(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SEARCHING FOR SHORTEST AND SAFEST PATHS ALONG OBSTACLE COMMON TANGENTS

by

Jerry Allen Crane
September 1991

Thesis Advisor: Yutaka Kanayama

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School — 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. — PROJECT NO. — TASK NO. — WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
SEARCHING FOR SHORTEST AND SAFEST PATHS ALONG OBSTACLE COMMON TANGENTS (U)

12. PERSONAL AUTHOR(S)
CRANE, JERRY ALLEN

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Master's Thesis	FROM $\overset{9/89}{05/89}$ TO $\overset{9/90}{03/90}$	SEPTEMBER 1990	

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Robotics, path planning, shortest paths, safe path planning, common tangents, convex subpolygons

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis describes a method for computing globally shortest paths for a point robot in a two-dimensional, orthogonal world composed of convex and concave polygons through the construction of obstacle common tangent visibility graphs. Visibility and intersection testing are based on the orientation of three or more points in the plane, and complex obstacle tangent visibility graphs are constructed using only these orientation relationships. Obstacle common tangents for convex and concave polygonal obstacles are implemented as a computational representation of locally shortest paths. A series of tangent sequences form global paths which equate to global path equivalence classes, effectively reducing the path finding problem to that of finding the shortest path in the path equivalence class. A simple and logical approach for processing concave polygons using convex subpolygons is implemented, allowing common tangent construction and path searching algorithms to process complex geometrical shapes in an efficient and symbolically unique fashion. Dijkstra's algorithm is implemented using heuristic control for optimal path searching. The framework for utilizing constant clearance strips for safe path planning along obstacle common tangents is presented but not fully implemented.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Yutaka Kanayama	22b. TELEPHONE *(Include Area Code)* (408) 646-2095 — 22c. OFFICE SYMBOL CS/Ka

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SEARCHING FOR SHORTEST AND SAFEST PATHS ALONG OBSTACLE COMMON TANGENTS

by
Jerry Allen Crane
Major, United States Army
B.S., USMA 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1991

Author: [illegible]
Jerry Allen Crane

Approved By: Yutaka Kanayama
Yutaka Kanayama , Thesis Advisor

[illegible]
Yuh-jeng Lee, Second Reader

[illegible]
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

This thesis describes a method for computing globally shortest paths for a point robot in a two-dimensional, orthogonal world composed of convex and concave polygons through the construction of obstacle common tangent visibility graphs. Visibility and intersection testing are based on the orientation of three or more points in the plane, and complex obstacle tangent visibility graphs are constructed using only these orientation relationships. Obstacle common tangents for convex and concave polygonal obstacles are implemented as a computational representation of locally shortest paths. A series of tangent sequences form global paths which equate to global path equivalence classes, effectively reducing the path finding problem to that of finding the shortest path in the path equivalence class. A simple and logical approach for processing concave polygons using convex subpolygons is implemented, allowing common tangent construction and path searching algorithms to process complex geometrical shapes in an efficient and symbolically unique fashion. Dijkstra's algorithm is implemented using heuristic control for optimal path searching. The framework for utilizing constant clearance strips for safe path planning along obstacle common tangents is presented but not fully implemented.

TABLE OF CONTENTS

I. INTRODUCTION

A. OBJECTIVES

The primary objective of the work in this thesis was to develop an application program implementing the theoretical work being done by Professor Yutaka Kanayama, U. S. Naval Postgraduate School, Monterey California, in searching for shortest and safest paths among obstacle common tangents. Obstacle common tangents are one form of representing path equivalence classes, as well as providing a representation of locally shortest paths.

The implementing this work equates to determining minimum cost paths for a robot moving in an environment that imposes certain geometric constraints. The main objectives of this thesis are the following:

1) Develop a two-dimensional world model capable of representing convex and concave polygons.
2) Construct a world tangent visibility graph providing symbolic representation of all common tangent situations.
3) Implement a searching method for the tangent vidibility graph using heuristic control to locate shortest paths.

B. BACKGROUND

We are currently employing an autonomous, program controlled, mobile robot Yamabico 11, to test and evaluate scientific and engineering problems related to robot guidance, navigation, and sensor integration (sonar and vision sensors and processors). Yamabico 11 operates using the Model-based Mobile robot Language (MML), which is a hardware independent standard language embedded in C that simplifies the programming of the robot [Kanayama.1988, p. 1]. A path in MML is specidfied using a series of robot postures. A single posture is a triple (x, y, θ), where (x, y) is the position and θ is the orientation of the robot in the global Cartesian coordinate system. A posture simply represents three degrees of freedom in the two dimensional plane of the vehicle.

The current method of generating path data for Yamabico 11 is through manually developing the series of postures required for movement of the robot from one point to another. Logically, the process of generating the necessary postures for Yamabico 11 to follow would be far easier if we could develop a path planning application which, when provided a world model, necessary start and goal points, and the robot's minimum clearance requirements, could provide the necessary robot postures to complete the desired movement without human intervention. Thus, the theoretical work on development of an efficient method for finding optimal safe paths for Yamabico 11 unfolded.

C. THESIS ORGANIZATION

The concepts which are introduced and applied in this thesis deal with finding globally shortest paths through a two-dimensional, orthogonal world, represented as distinct convex and concave polygons. The first three chapters of this work detail much of the as yet unpublished work of Professor Yutaka Kanayama on searching for shortest and safest paths.

The mathematical foundations for path planning using obstacle common tangents is presented in considerable detail in Chapter II, beginning with the simple geometric relationships between points, polygons and lines in the two-dimensional plane. The construction and identification of obstacle common tangents is explained, and we introduce the convex subpolygon; a very simple and elegant method we use to represent and process concave polygons, allowing all obstacles to be processed in a standardized symbolic manner.

Chapter III addresses the efficient visibility testing between two points, interpretation of tangent sequences, and links paths along xommon tangents to path equivalence classes and locally shortest paths. We conclude the chapter by outlining the necessary tools for developing efficient searching algorithms using pre-processed obstacle tangent visibility graphs. Chapter IV completes the theoretical background by describing the process of building constant clearance paths along obstacle common tangents, yeilding shortest safest paths for robot vehicles.

In Chapter V, the implementation of optimal path searching among obstacle common tangents is detailed. A working application is presented for constructing obstacle tangent visibility graphs, as well as path finding through the tangent visibility graph using heuristic control of a modified Dijkstra search. A detailed account of this implementation in Common Lisp is given, as well as the graphical screen displays showing the shortest path searching process. Chapter VI addresses the contributions of this work, and outlines the areas where additional research effort is needed.

II. MATHEMATICAL BASIS FOR COMMON TANGENTS

A. POINT REPRESENTATION

The path planning concepts of this application deal with moving a robot of two dimensions through planar space. We therefore confine our representation to a two dimensional plane where we can represent any point by its global Cartesian coordinates. Let

$$\mathbf{p_1 = (x_1, y_1) \text{ and } p_2 = (x_2, y_2)} \qquad (1)$$

be two points given by their global coordinates and $(x_1, y_1) \neq (x_2, y_2)$ (Figure 1).

B. ORIENTATION

An *orientation* from p_1 to p_2 is measured from the positive x-axis orientation in a counter-clockwise direction. Evaluation of this orientation can be accomplished using the inverse tangent function as follows:

$$\mathbf{\textit{orientation}(p_1, p_2) = tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)} \qquad (2)$$

and orientation values range $[-\pi, \pi]$.

Consider the two points p_1=(2, 1) and p_2=(3, 2) (Figure 2),

$$\mathbf{orientation(p_1, p_2) = tan^{-1}(\ 1\ ) = \frac{\pi}{4}}$$

$$\mathbf{orientation(p_2, p_1) = tan^{-1}(\ 1\ ) = \frac{\pi}{4}.}$$

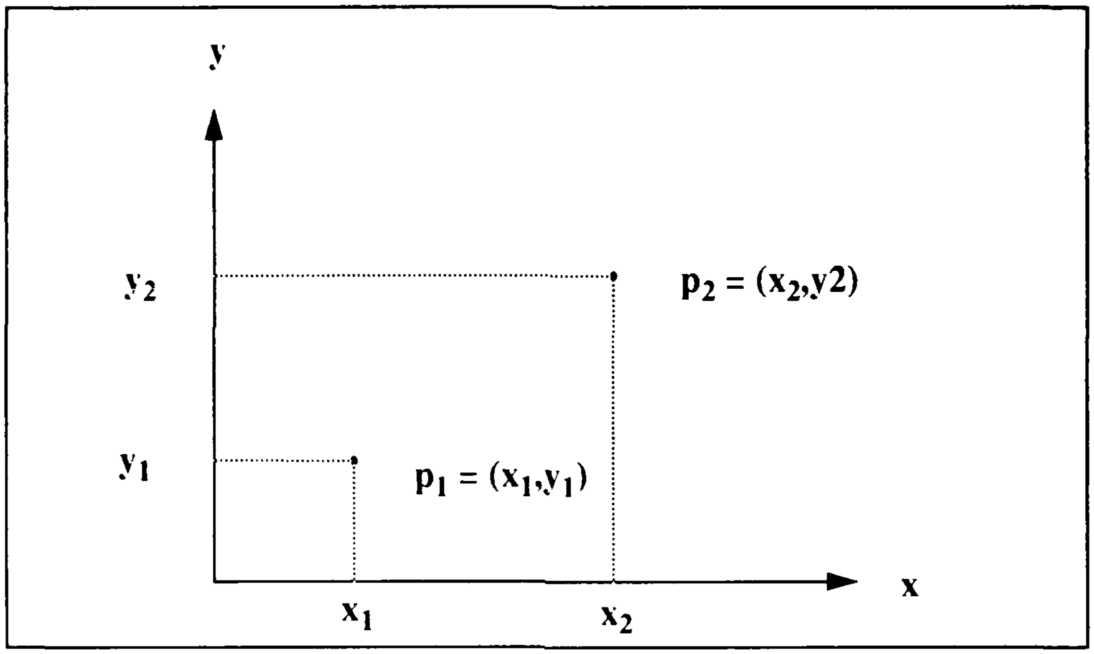


Figure 1 - Cartesian Coordinate System

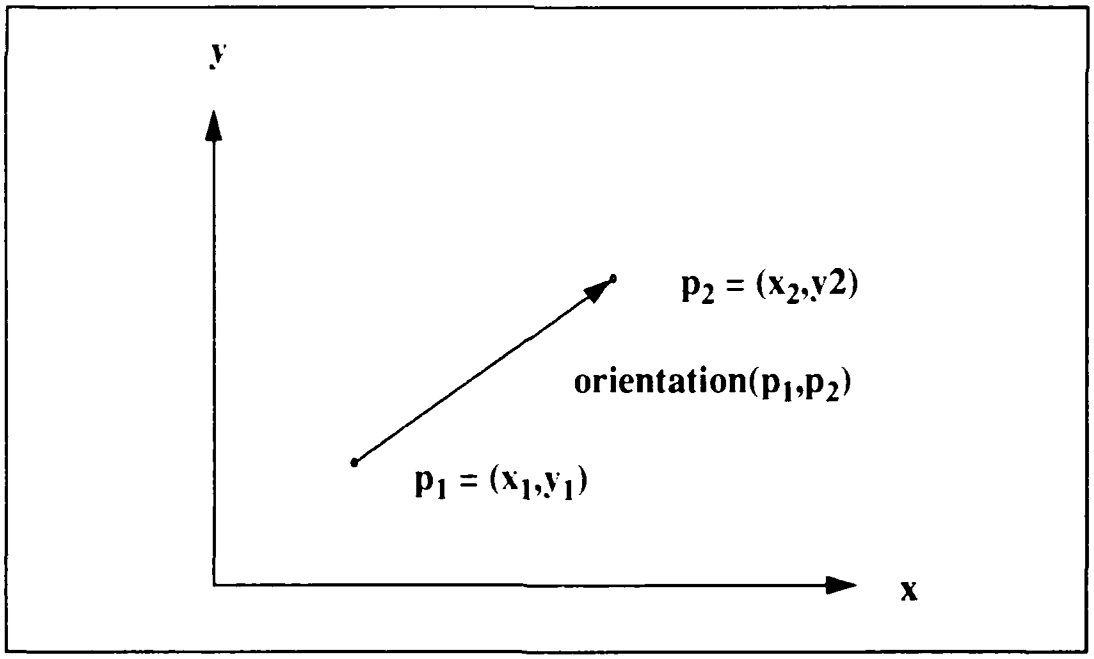


Figure 2 - Orientation Between Two Points

This method has two serious shortcomings. First, the orientation function does not yield a distinct value based on the ordering of a given pair of distinct points. What we desire is

$$\textbf{\textit{orientation}}(p_1, p_2) = -\frac{3\pi}{4} \neq \textbf{\textit{orientation}}(p_2, p_1) = \frac{\pi}{4}.$$

However, the *orientation* function defined in Equation 2 yields

$$\textbf{\textit{orientation}}(p_1, p_2) = \textbf{\textit{orientation}}(p_2, p_1) = \frac{\pi}{4}.$$

for all point pairs considered. The second problem is that Equation 2 is undefined when $x_1 = x_2$. The solution to these problems is the use of the inverse-tangent function using two arguments (*atan2*). The *atan2* function is implemented in most modern programming languages. We redefine ***orientation*** as

$$\textbf{\textit{orientation}}(p_1,p_2) = \textbf{\textit{atan2}}(y_2-y_1, x_2-x_1) \qquad \textbf{(3)}$$

The range of values is $[-\pi, \pi]$. This definition of *orientation* will function as desired over all pairs of points and we are assured that

$$\textbf{\textit{orientation}}(p_1,p_2) - \textbf{\textit{orientation}}(p_2,p_1) = +/-\pi. \qquad \textbf{(4)}$$

C. NORMALIZATION OF ORIENTATIONS

Restricting an orientation to the range $[-\pi, \pi]$ thru normalization is not always necessary nor desirable. Assume α represents the orientation of a robot or other vehicle which may execute an indefinite number of orientations along a path (Figure 3). In this case, the current orientation contains information regarding the cumulative number of rotations of the vehicle if it has not been normalized. The fact that one or more full or partial rotations have occurred can be an element of essential global information. There appears to be no situation in which these unnormalized orientations cause a problem.

In dealing with angles, normalization of orientations is generally required. For any orientations α_1 and α_2, we write $\alpha_1 \equiv \alpha_2$ if there exists an integer n such that $\alpha_1 - \alpha_2 = 2n\pi$. Consider an angle β from an orientation α_1 to another, α_2 (Figure 4). We have already indicated that the orientations are not normalized. This leads to the conclusion that the range of the angle β is unbounded since it may include an undetermined number of rotations. The desired result is the normalized angle, β, free of any rotational data.

The function *normalize* is defined as

$$\mathit{normalize}(\beta) \equiv \beta \tag{5}$$

and

$$\mathit{normalize}(\beta) \text{ element of } [-\pi, \pi]. \tag{6}$$

As an example, consider *normalize*$(2.5\pi) = 0.5\pi$ (Figure 5a) and *normalize*$(3.5\pi) = -0.5\pi$ (Figure 5b). Using this normalization function, we define an order $<$ among orientations such that for any orientations α_1 and α_2,

$$\alpha_1 < \alpha_2 \tag{7}$$

if and only if

$$\mathit{normalize}(\alpha_1 - \alpha_2) > 0. \tag{8}$$

For instance, $-\frac{\pi}{4} < \frac{\pi}{4}$ and $\frac{3\pi}{4} < -\frac{3\pi}{4}$. We can write $\alpha_1 \leq \alpha_2$ if *normalize*$(\alpha_2 - \alpha_1) \geq 0$.

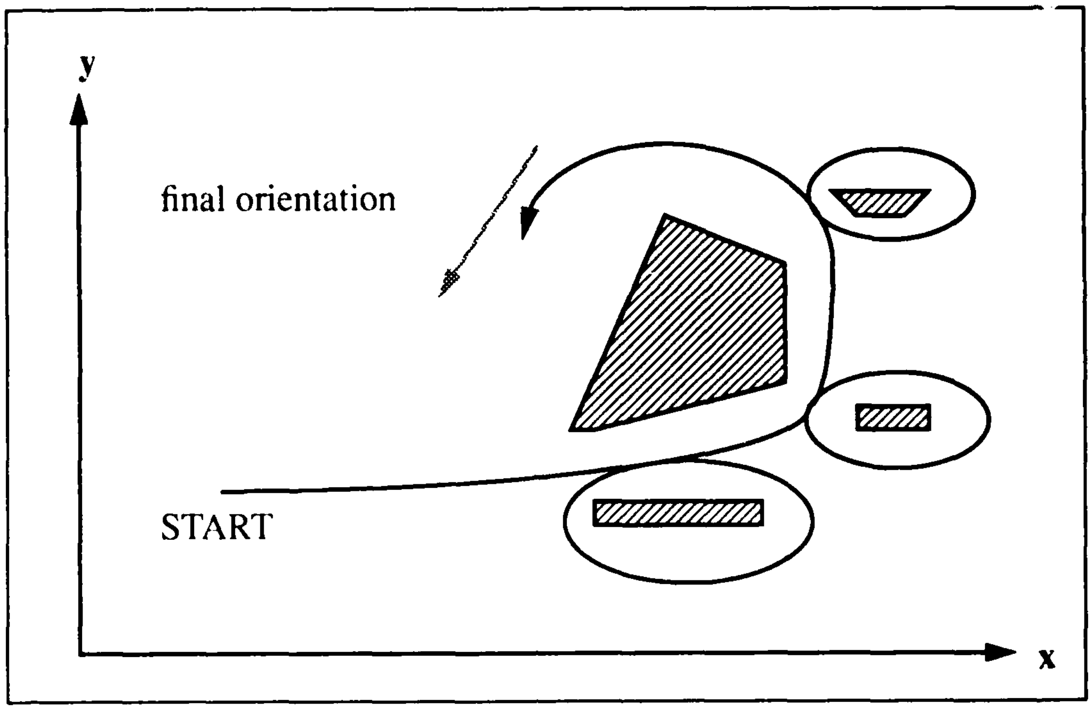


Figure 3 - Normalization of Robot Orientation

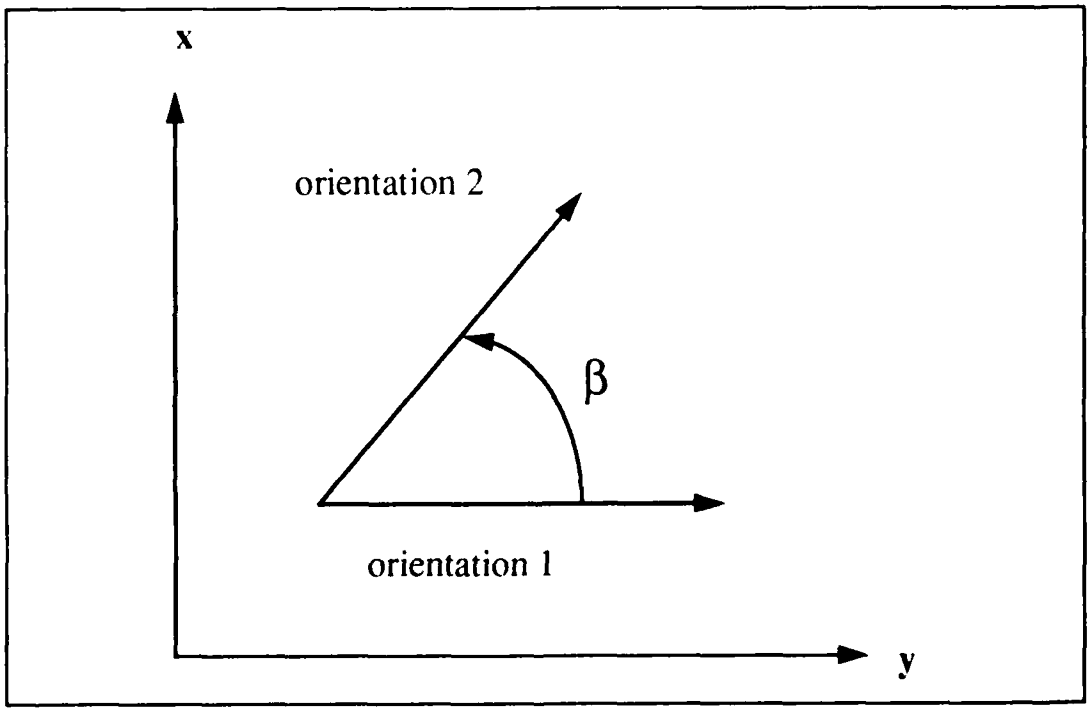


Figure 4 - Angle Between Two Orientations

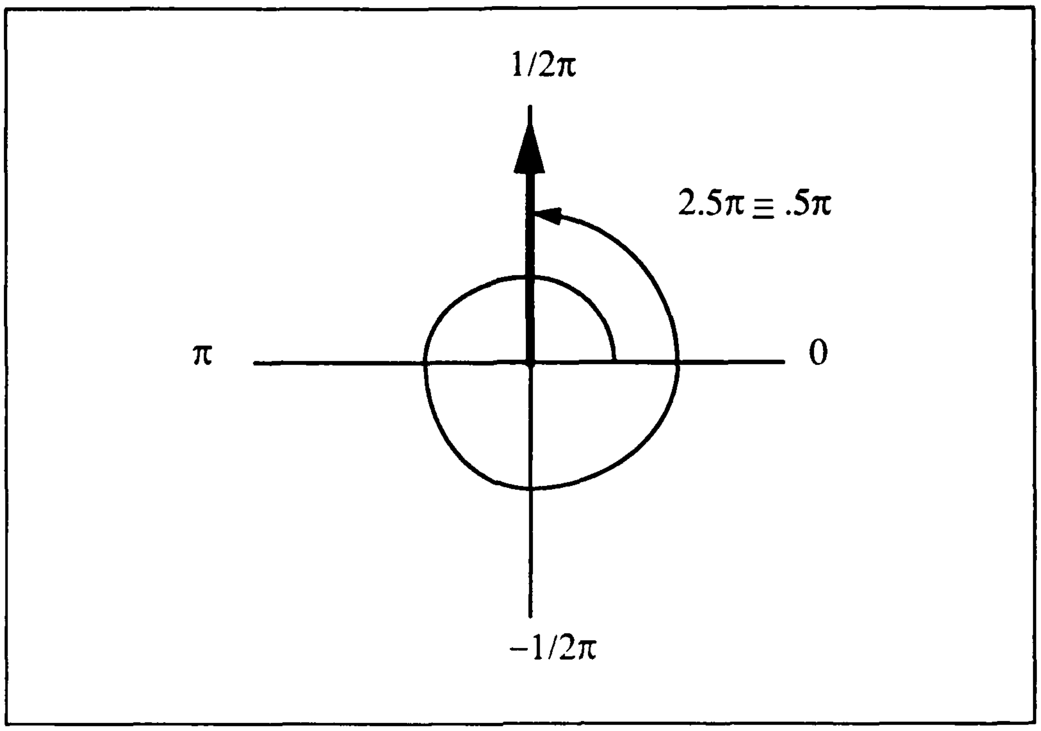


Figure 5a - Equivalent Orientations After Normalization

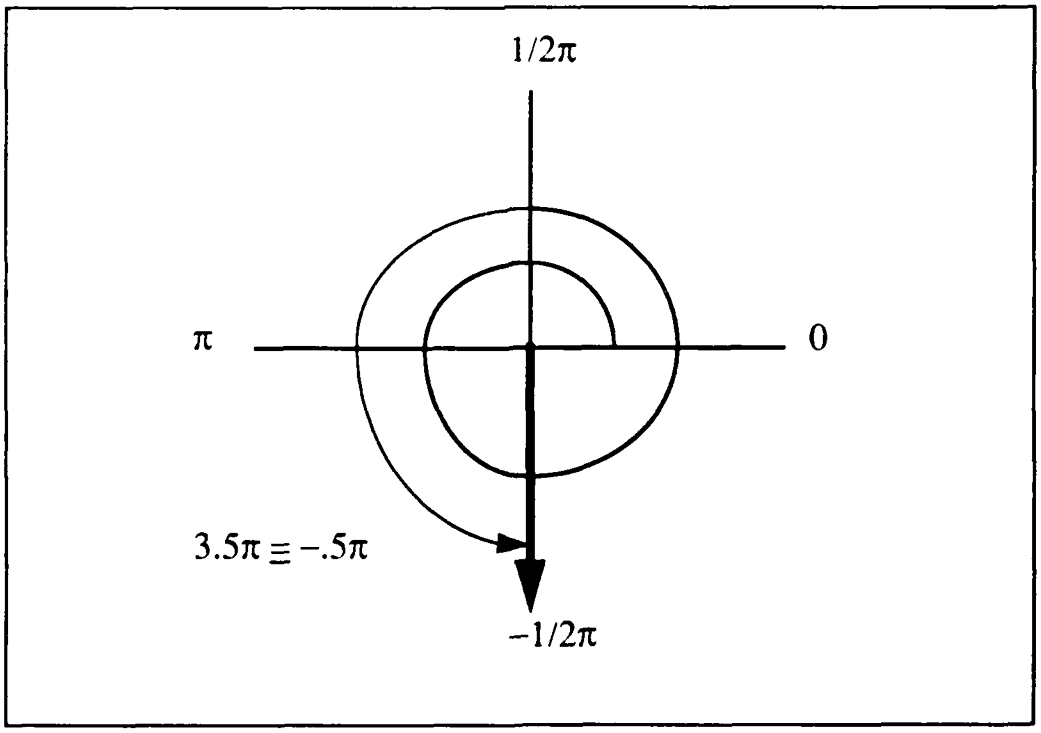


Figure 5b - Equivalent Orientations After Normalization

Consider a triple(p_1,p_2,p_3) of distinct points. The ordering of these points is said to be counter-clockwise (Figure 6), written *ccw*(p_1,p_2,p_3), if and only if

$$\textbf{\textit{orientation}}(p_1,p_2) < \textbf{\textit{orientation}}\ (p_2,p_3). \tag{9}$$

It is said to be clockwise, written as *cw*(p_1,p_2,p_3) (Figure 7), if and only if

$$\textbf{\textit{orientation}}(p_2,p_3) < \textbf{\textit{orientation}}\ (p_1,p_2). \tag{10}$$

When the points lie on the same line, they are obviously collinear, and written as (p_1,p_2,p_3). In terms of the orientations of the point pairs, we can say

$$\begin{gathered} \textbf{\textit{orientation}}(p_1,p_2) \equiv \textbf{\textit{orientation}}\ (p_2,p_3) \\ \textbf{or} \\ \textbf{\textit{orientation}}(p_1,p_2) \equiv \textbf{\textit{orientation}}\ (p_2,p_3) + \pi. \end{gathered} \tag{11}$$

D. Placement of Three Points

Consider now a triple(p_1 p_2 p_3) of distinct points, where $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$. The three points can serve as vertices of a triangle, $\Delta p_1\ p_2\ p_3$ (Figure 8). Let us define $S(p_1, p_2, p_3)$ as

$$S(p_1, p_2, p_3) \equiv \begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} \tag{12}$$

$$= \tfrac{1}{2} [x_2\, y_3 + x_3\, y_1 + x_1\, y_1 - x_1\, y_3 - x_2\, y_1 - x_3\, y_2] \tag{13}$$

$$= \tfrac{1}{2} [(x_2 - x_1)\,(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)] \tag{14}$$

The absolute value of $S(p_1, p_2, p_3)$ equals the area of $\Delta p_1,p_2,p_3$. Using Equation 12, consider the following values of S for $\Delta p_1,p_2,p_3$:

(1) The area of $\Delta p_1,p_2,p_3$ is equal to $S(p_1, p_2, p_3)$ if the ordering of p_1 p_2 p_3 is in a counter-clockwise orientation: written *ccw*(p_1 p_2 p_3).

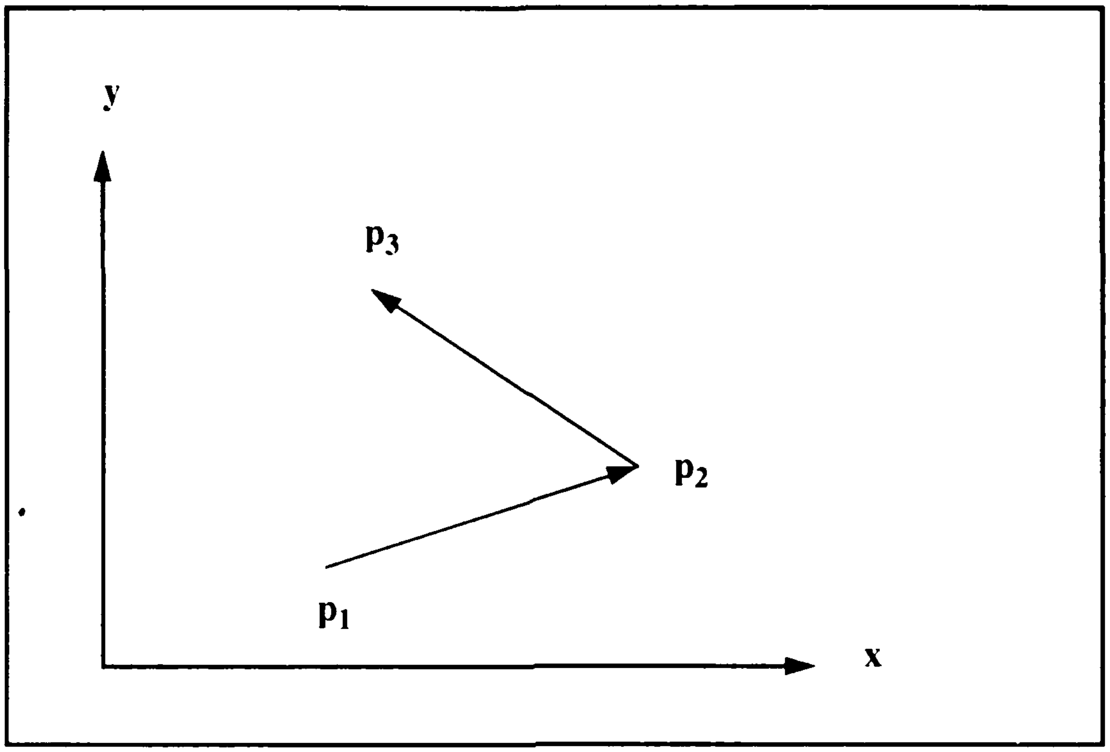


Figure 6 - Counter-Clockwise Ordering

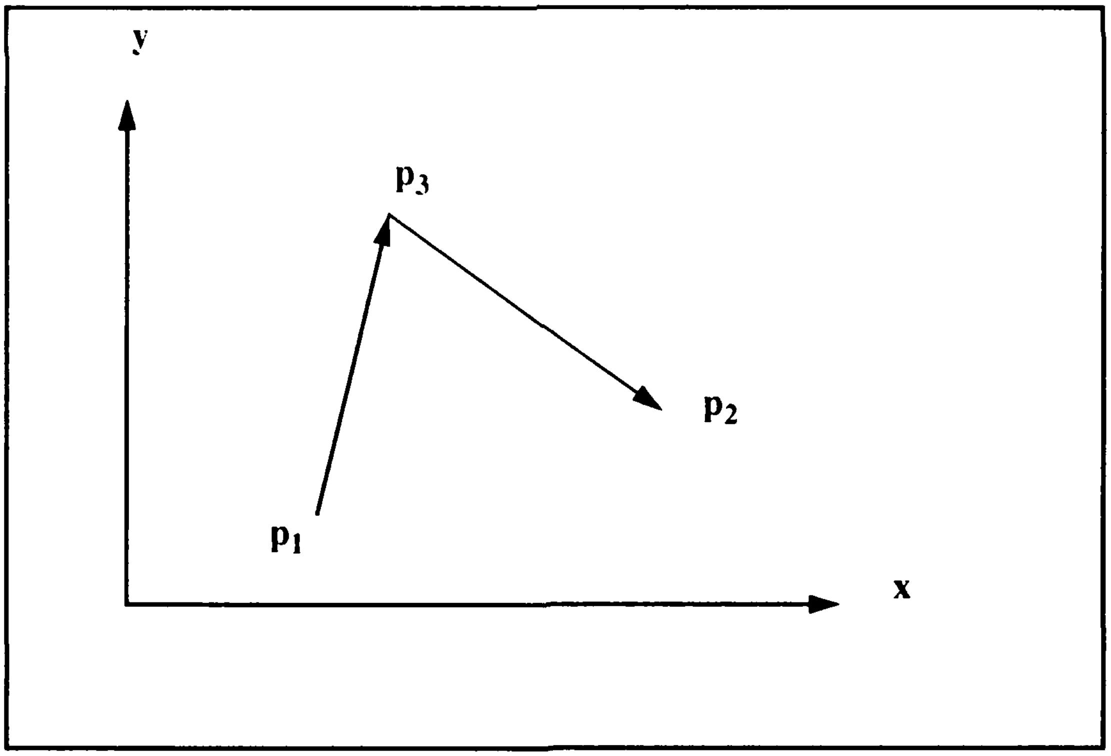


Figure 7 - Clockwise Ordering

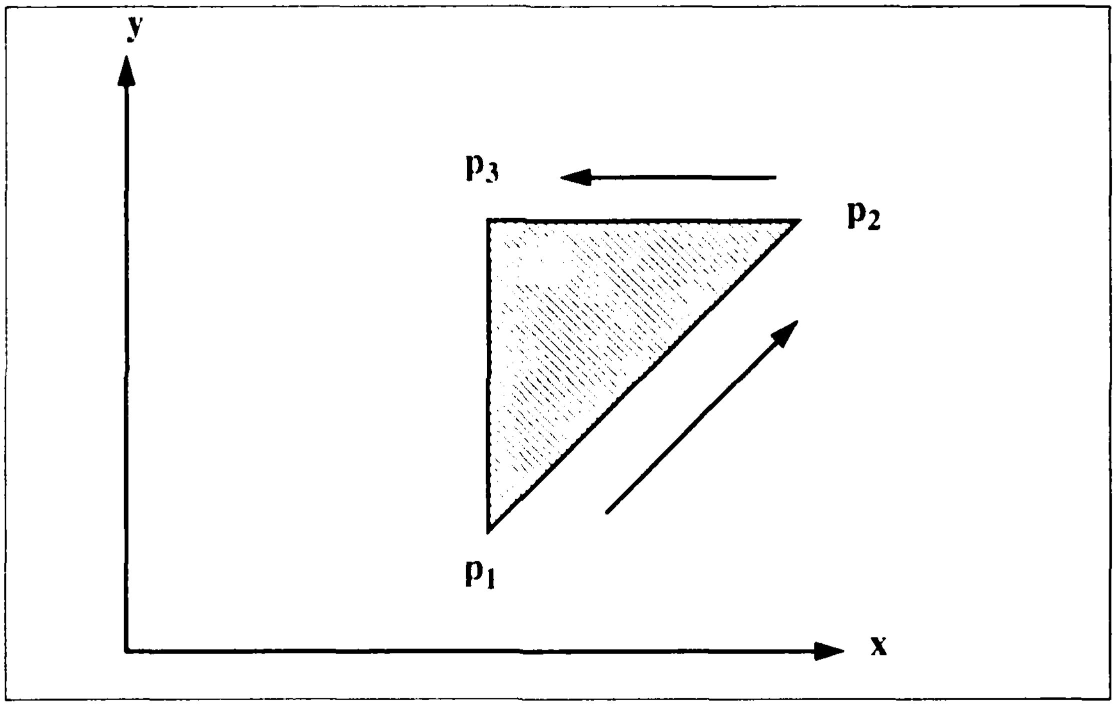


Figure 8 - Placement of Three Points

(2) The area of $\Delta p_1, p_2, p_3$ is equal to $-S(p_1, p_2, p_3)$ if the ordering of $p_1\ p_2\ p_3$ is in a clockwise orientation; written $cw(p_1\ p_2\ p_3)$.

(3) The area of $\Delta p_1, p_2, p_3$ is equal to $S(p_1, p_2, p_3) = 0$, regardless of the orientation, if $collinear(p_1\ p_2\ p_3)$.

We can now define a function, order, which returns the value, positive/negative or zero of $S(p_1, p_2, p_3)$ as

$$order(p_1\ p_2\ p_3) \equiv sign(S(p_1\ p_2\ p_3)) \qquad (15)$$

where

$$sign(x) = \begin{cases} -1 \text{ if } x < 0 \\ 0 \text{ if } x = 0 \\ 1 \text{ if } x > 0 \end{cases} \qquad (16)$$

and for any triple of points $p_1\ p_2\ p_3$, we can further state that

$$order(p_1\ p_2\ p_3) = \begin{cases} -1 \text{ if } cw(p_1\ p_2\ p_3) \\ 0 \text{ if } collinear(p_1\ p_2\ p_3) \\ 1 \text{ if } ccw(p_1\ p_2\ p_3) \end{cases} \qquad (17)$$

Given an ordered triple of three distinct points, we have outlined two tests to determine the counter-clockwise or clockwise placement of the three points with respect to each other. Equations 8, 9 and 10 can be used after calculating the orientations of the individual points, or Equations 15, 16 and 17 are available. Both testing methods are effective the choice of one over the other depends on the nature of the calculations. The simple concept of testing orientation and point placement are the foundation for the methods outlined in the following chapters.

E. Crossing Tests for Line Segments

We can extend the idea of the placement and orientation of three points to develop efficient crossing tests for open and closed line segments. The computational cost of these tests are constant, built on simple multiplication and addition. An open (or closed) line segment can be defined by two distinct end points. Further, two segments are said to cross if they share exactly one point in common. Consider two line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ each defined by their respective end points. Two segments cross under the following circumstances:

(1) $\overline{p_1p_2}$ and $\overline{p_3p_4}$, both open segments, intersect at a point p, and p is not one of the four end points (Figure 9), or,

(2) $\overline{p_1p_2}$ and $\overline{p_3p_4}$, closed segments, have one or two end point of their end points as the crossing point (Figure 10).

In terms of point orientations, a necessary and sufficient condition for two open segments to cross is

$$\begin{gathered}[\mathit{order}(p_1p_2p_3) = -\,\mathit{order}(p_1p_2p_4) \neq 0] \\ \textbf{and} \\ [\mathit{order}(p_3p_4p_1) = -\,\mathit{order}(p_3p_4p_2) \neq 0]\end{gathered} \tag{18}$$

A necessary and sufficient condition for two closed segments to cross is

$$\begin{gathered}[\mathit{order}(p_1p_2p_3) \neq \mathit{order}(p_1p_2p_4)] \\ \textbf{and} \\ [\mathit{order}(p_3p_4p_1) \neq \mathit{order}(p_3p_4p_2)]\end{gathered} \tag{19}$$

The above crossing tests can be used as components of more complex tests like visibility testing and line segment intersection tests. It should be noted that if two open segments cross, so do the corresponding closed segments. Its converse, however, is not true.

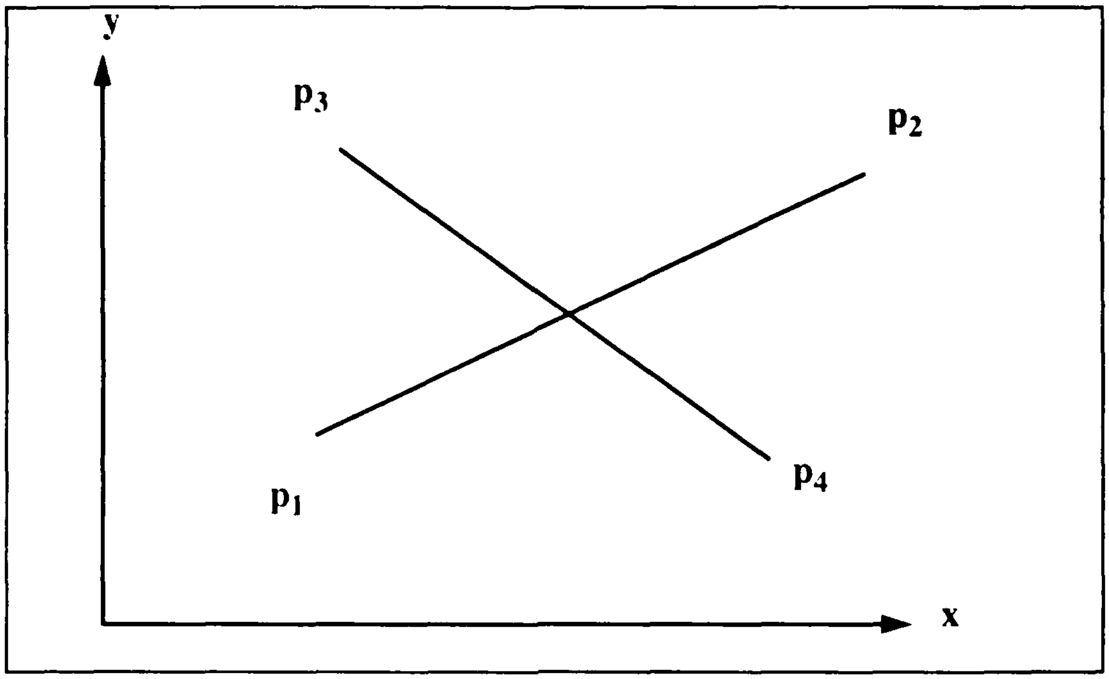


Figure 9 - Crossing of Two Open Segments

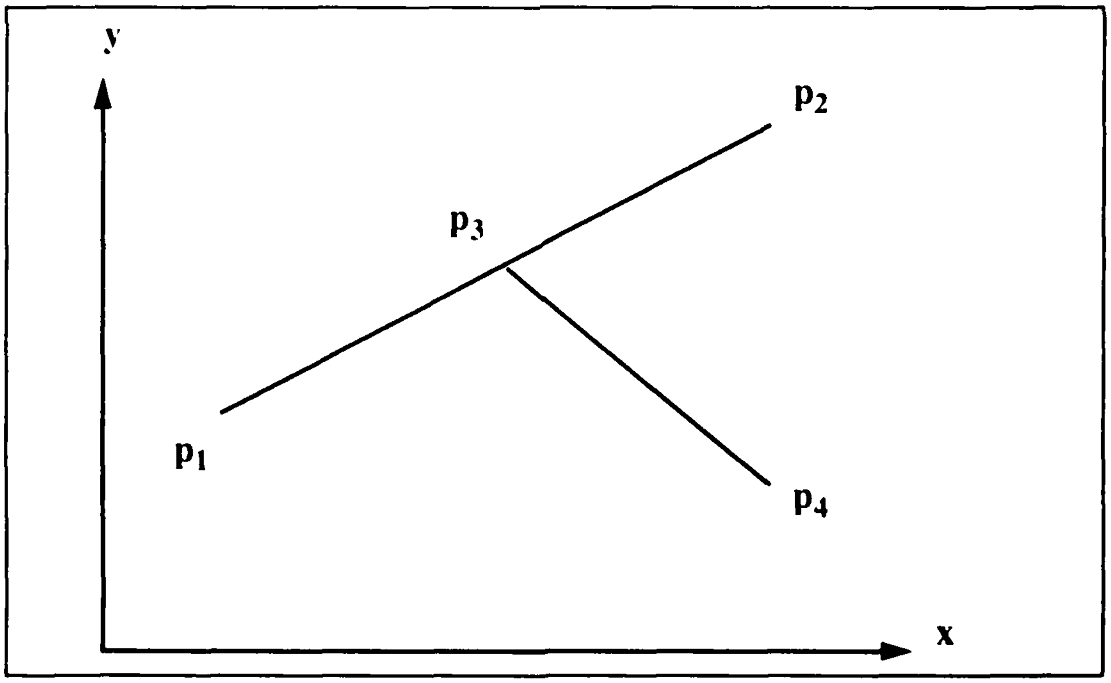


Figure 10 - Crossing of Two Open or Closed Segments

F. REPRESENTATION OF POLYGONS

We have outlined crossing tests for line segments, but still lack a method for modeling the two dimensional environment where our robot or vehicle will be required to navigate. The two dimensional model will be constructed from polygons, both convex and concave, as defined in this section.

Let V be a set of n ($n \geq 3$) distinct points, v_0 a point in V, and *next* be a function such that $V \rightarrow V$. A triple B = (V, v_0, *next*) is said to be a polygon if the following conditions hold:

$$\{next^0(v_0),...,next^{n-1}(v_0)\} = V$$
and
$$next^n(v_0) = v_0 \qquad \textbf{(20)}$$

where a function $next^n$ is defined as

$$next^n \equiv \begin{cases} next^0(v) = v \\ next^{i+1}(v) = \text{next}\ (next^i(v)) \quad \textbf{for any } i \geq 0 \end{cases} \qquad \textbf{(21)}$$

for any v element V. Additionally, we can say that for any v and v', elements of V,

(1) the two segments $\overline{v\ next(v)}$ and $\overline{v'\ next(v')}$ do not cross, and,

(2) that for any v element V, a triple of points (v, *next*(v), *next*(*next*(v))) are not collinear.

The following properties concerning the function next are proposed, allowing us to develop and define the inverse function of next, previous.

(1) If B = (V, p_0, *next*) is a polygon, *next* is a one-to-one mapping of the vertices of B.

(2) For any v element V with n vertices, $next^n(v) = v$.

Thus, if *next*(v) = u, then *previous*(u) = v. We now define v_i, which allows reference to be made to any specific point in describing a polygon, v_i, as

$$v_i \equiv \textit{next}^{\,i}(v_0) \text{ for all } i > 0 \tag{22}$$

A point v in a polygon is called a *vertex*. A segment $\overline{v\, \textit{next}(v)}$ of a polygon is called an (directed) *edge*. The set of all the closed edges of a polygon is called the *boundary* of the polygon. A polygon boundary forms a simple directed loop. A well known theorem by Jordan, known as Jordan's Curve Theorem, states that a polygon boundary divides a plane into two parts. The finite part of the plane divided by the boundary of a polygon is called its *inside*, the other part its *outside* (Figure 11). Also, the right side of a boundary, when traversing along the directed edges, is called *free space*. The left side is called *filled space* (Figure 12 and 13). It is within this free space that a robot can exist, and with which the path planning process is concerned.

In a polygon B, we define the orientation γ_i of the *i*th edge

$$\gamma_i \equiv \textit{orientation}(v_i, v_{i+1}) \tag{23}$$

and the angle δ_i as the external angle of the *i*th vertex (Figure 14), v_i, as

$$\delta_i \equiv \textit{normalize}(\gamma_{i+1} - \gamma_i) \tag{24}$$

A vertex on a polygon B is said to be convex is its external angle is positive, i.e.,

$$\textit{ccw}(v, \textit{next}(v), \textit{next}(\textit{next}(v))). \tag{25}$$

A polygon B is said to be *convex* if and only if all the vertices in B are convex. If a polygon is not *convex*, it is said to be *concave*. For any polygon B,

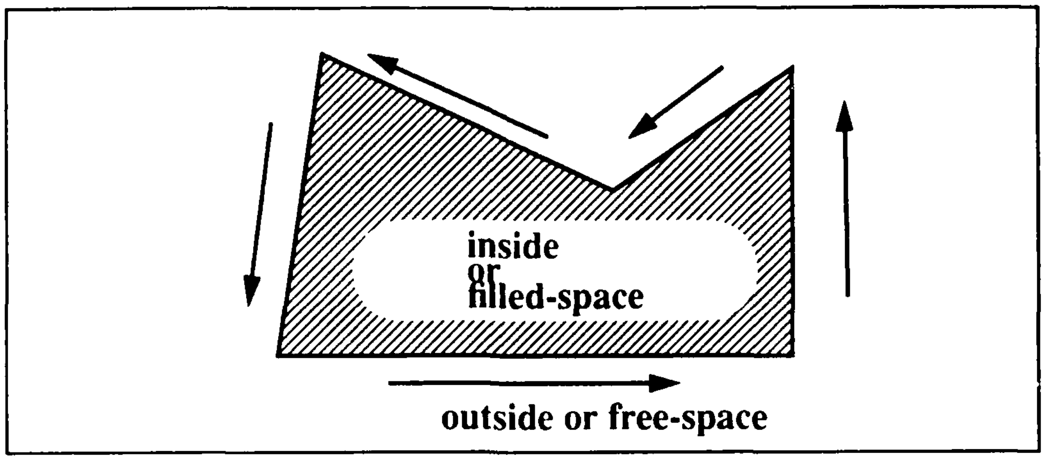


Figure 11 - Polygon Inside/Outside

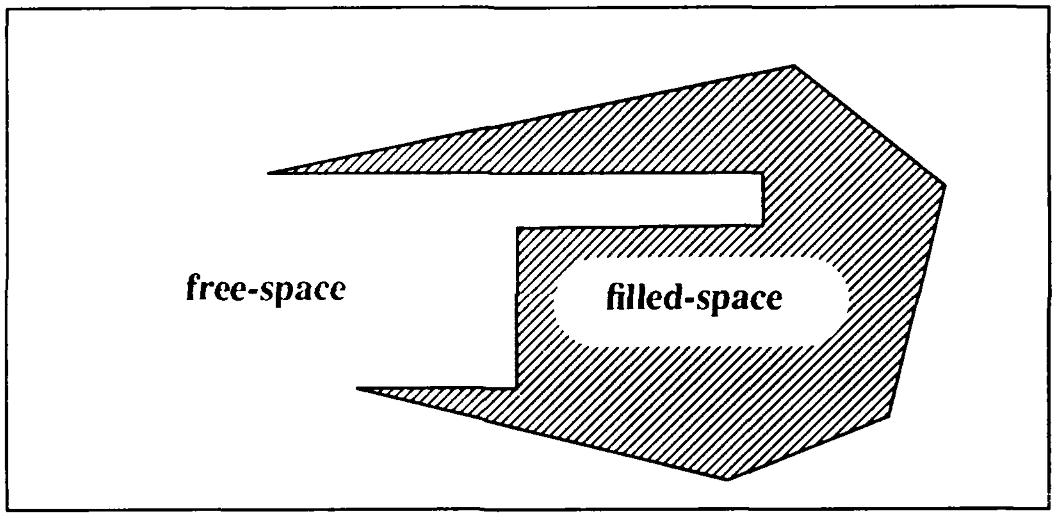


Figure 12 - Filled and Free-space Solid Polygon

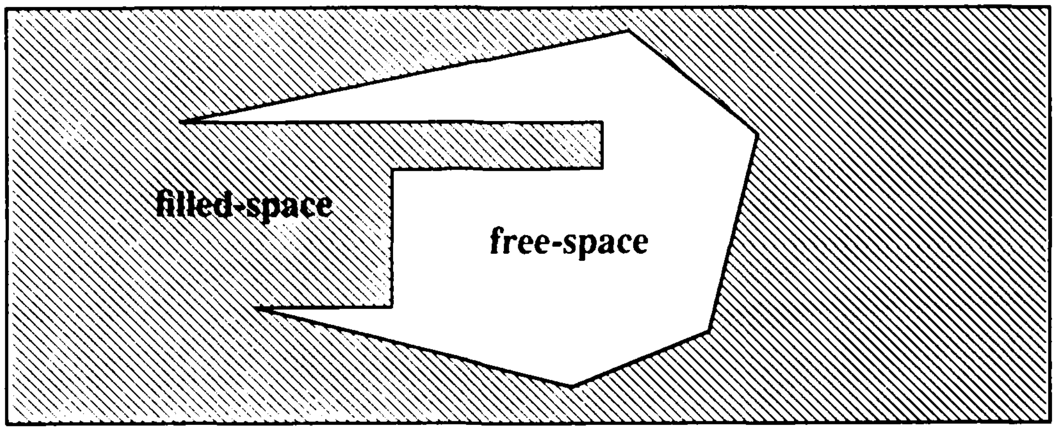


Figure 13 - Filled and Free-space Hollow Polygon

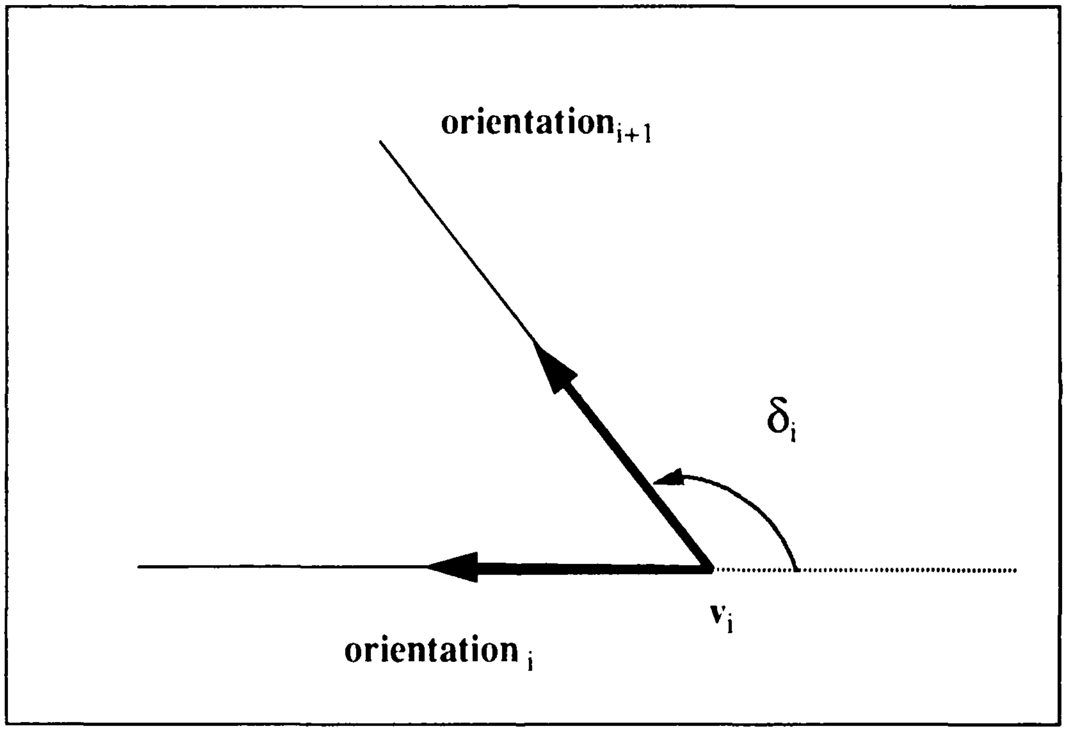


Figure 14 - Vertex External Angle

$$\sum_{i=0}^{n-1} \delta_i = \pm 2\pi$$

If $\sum \delta_i = 2\pi$, the polygon is called *normal*, and if $\sum \delta_i = -2\pi$, B is called *inverted.* We can further propose that

(1) If a polygon is convex, it is normal.
(2) If a polygon is inverted, it is concave.

G. TANGENTS OF POLYGONS

Movement through a two dimensional environment with obstacles modeled by polygons requires paths which can avoid obstacles while presenting the shortest, most direct paths through the obstacles. The most direct and shortest path from a point around an obstacle (polygon) will lie on a line beginning at that point and tangent to the obstacle to be avoided. The following section defines the "*plus*" and "*minus*" tangent modes (counter-clockwise and clockwise tangents) to polygons and details the necessary and sufficient conditions for these tangents to exist.

Let B be a polygon, and ℓ a directed line or ray existing in B's free space. ℓ is said to be tangent to B if a vertex of B, or a pair of adjacent vertices of B, are on ℓ (Figure 15). The vertex (or adjacent vertices) where ℓ and B intersect is called the *osculating vertex* (Figure 16). Given a point p in B's free space, it is possible to construct exactly two tangents from p to B if B is convex (Figure 17). If B is a concave polygon, it may be possible to construct multiple tangents from p to B, depending on the positioning of p with respect to B and its convex vertices (Figure 18).

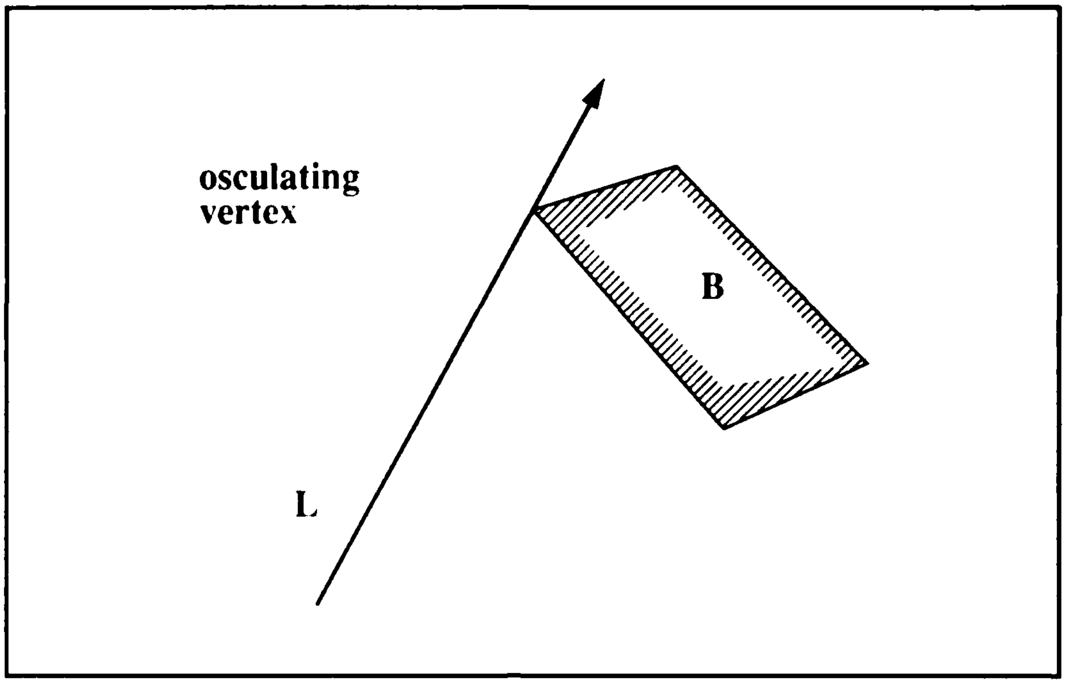


Figure 15 -Tangent with a Single Osculating Vertex

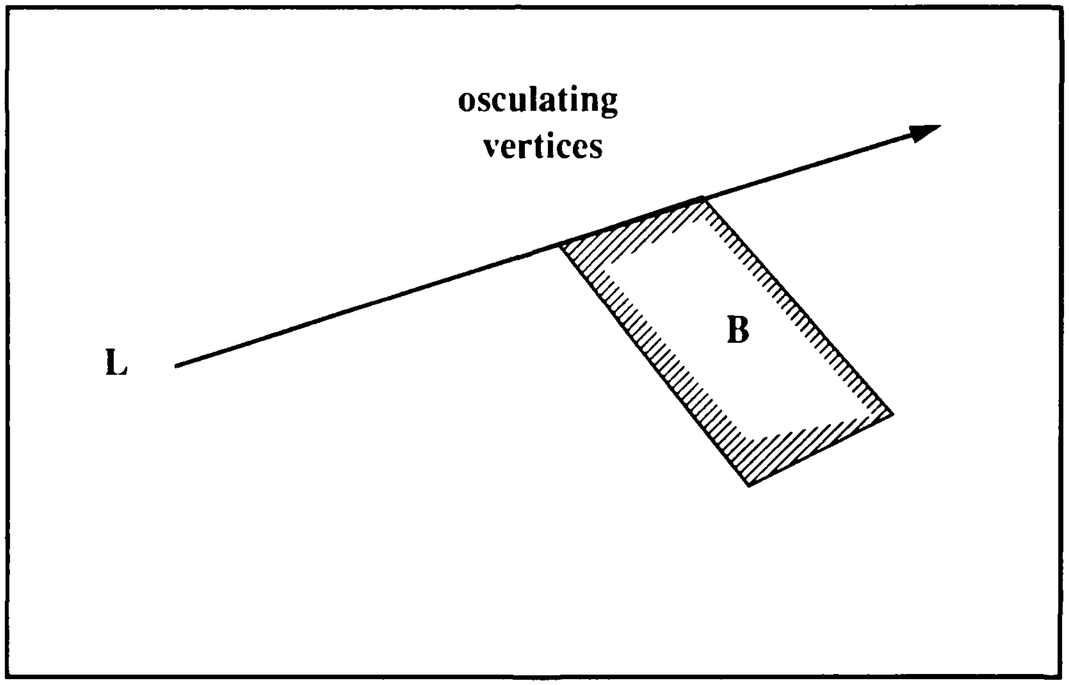


Figure 16 -Tangent Osculating on TwoVertices

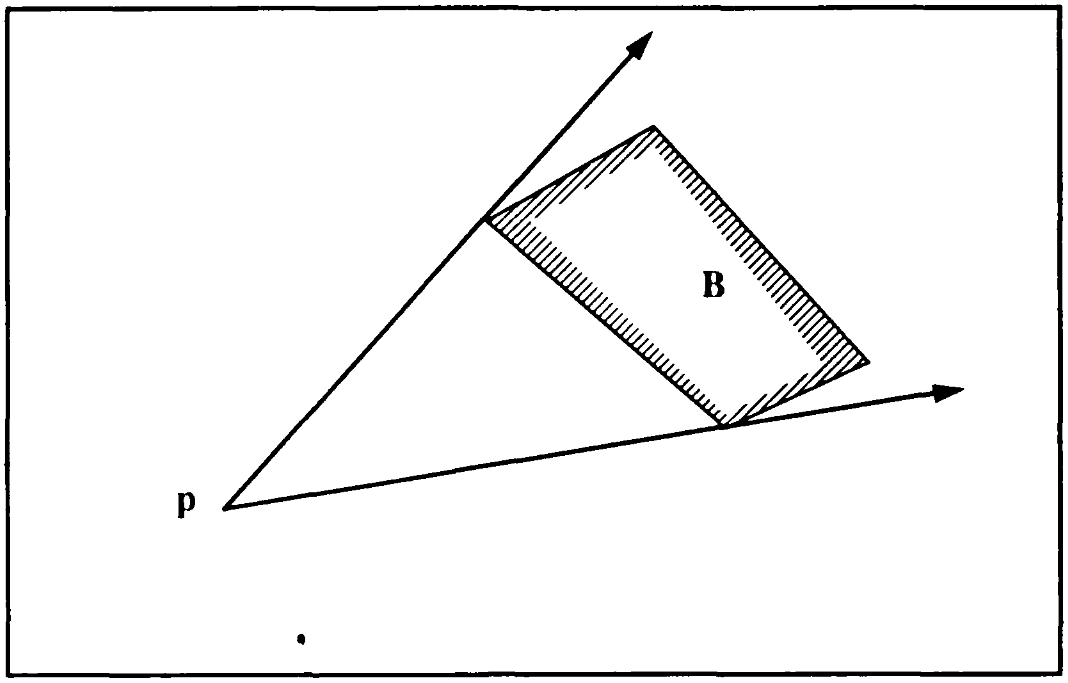


Figure 17 - Tangents From a Point to a Convex Polygon

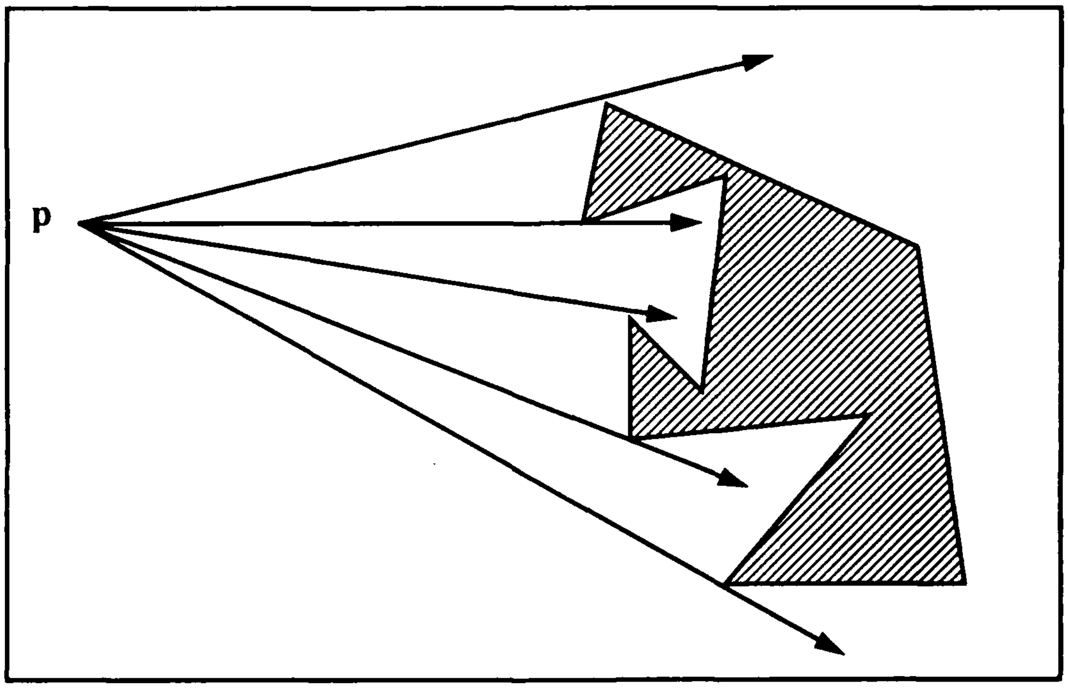


Figure 18 - Multiple Tangents From a Point to a Concave Polygon

The line ℓ is said to be a plus or counter clockwise tangent to B (Figure 19) if

(1) a vertex v of B is on ℓ and its adjacent vertices *next*(v) and *previous*(v) are on the left side of ℓ, or,

(2) two adjacent vertices v and u of B are on ℓ and their adjacent vertices, *next*(v) and *previous*(u) or *previous*(v) *next*(u), are on the left side of ℓ.

ℓ is said to be a minus or clockwise tangent to B (Figure 20) if

(1) a vertex v of B is on ℓ and its adjacent vertices *next*(v) and *previous*(v) are on the right side of ℓ, or,

(2) two adjacent vertices v and u of B are on ℓ and their adjacent vertices, *next*(v) and *previous*(u) or *previous*(v) *next*(u), are on the right side of ℓ.

Plus and minus tangent modes can be defined in terms of the orientations of the point p, the osculating vertex of B, and its adjacent vertices. If v is an osculating point of a plus tangent from p to B, then

$$\textit{\textbf{ccw}}\textbf{(p,v,}\textit{\textbf{previous}}\textbf{(v)) and }\textit{\textbf{ccw}}\textbf{(p,v,}\textit{\textbf{next}}\textbf{(v)).} \qquad \textbf{(26)}$$

If v and *next*(v) are both osculating points of a *plus tangent* from p to B then

$$\begin{gathered}\textit{\textbf{ccw}}\textbf{(p,v,}\textit{\textbf{previous}}\textbf{(v)) and }\textit{\textbf{collinear}}\textbf{(p,v,}\textit{\textbf{next}}\textbf{(v))}\\ \textbf{and}\\ \textit{\textbf{ccw}}\textbf{(p,}\textit{\textbf{next}}\textbf{(v),}\textit{\textbf{next}}\textbf{(}\textit{\textbf{next}}\textbf{(v))).}\end{gathered} \qquad \textbf{(27)}$$

If v is an osculating point of a *minus tangent* from p to B, then

$$\textit{\textbf{cw}}\textbf{(p,v,}\textit{\textbf{previous}}\textbf{(v)) and }\textit{\textbf{cw}}\textbf{(p,v,}\textit{\textbf{next}}\textbf{(v)).} \qquad \textbf{(28)}$$

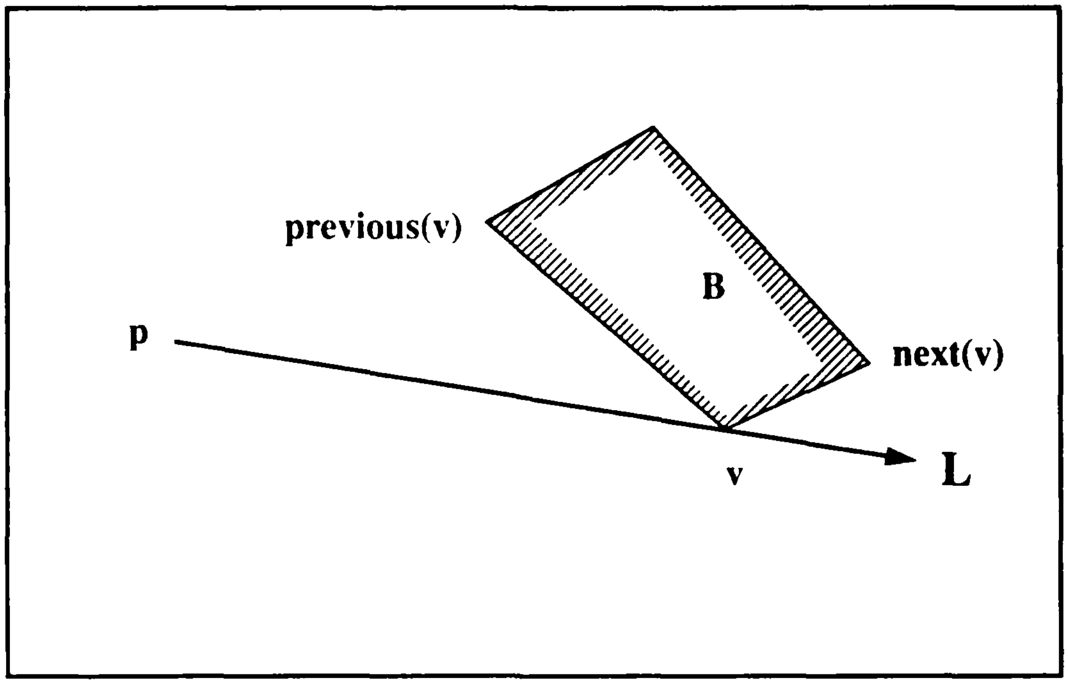


Figure 19 - Counter-Clockwise Tangent B+

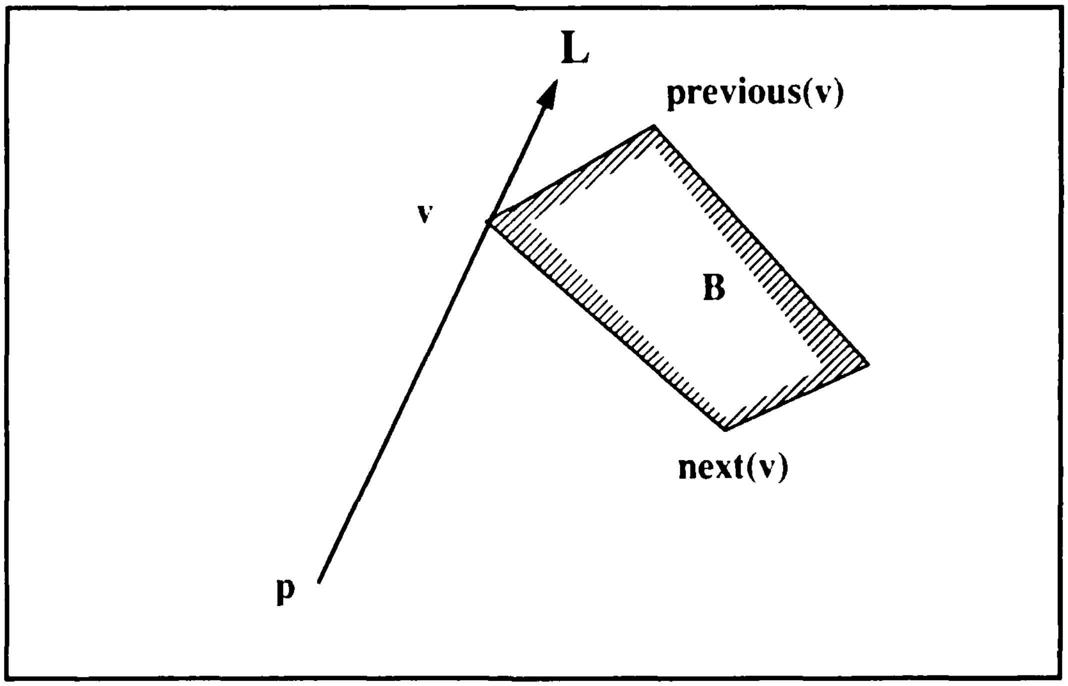


Figure 20 - Clockwise Tangent B-

If v and *next*(v) are both osculating points of a *minus tangent* from p to B then

cw(p,v,*previous*(v)) and *collinear*(p,v,*next*(v))

and

cw(p,*next*(v),*next*(*next*(v))). **(29)**

Similar propositions for plus and minus tangent modes from a polygon B to a point p in its free space are determined in a like fashion by reversing orientations (Figure 21). If v is an osculating point of a *plus tangent* from B to p, then

cw(p,v,*previous*(v)) and cw(p,v,*next*(v)). **(30)**

If v and *next*(v) are both osculating points of a *plus tangent* from B to p then

cw(p, v, *previous*(v)) and *collinear*(p, v, *next*(v))

and

cw(p, *next*(v), *next*(*next*(v))). **(31)**

If v is an osculating point of a *minus tangent* from B to p, then

ccw(p, v, previous(v)) and ccw(p, v, next(v)). **(32)**

If v and *next*(v) are both osculating points of a *minus tangent* from B to p then

ccw(p, v, *previous*(v)) and *collinear*(p, v, *next*(v))

and

ccw(p, *next*(v), *next*(*next*(v))). **(33)**

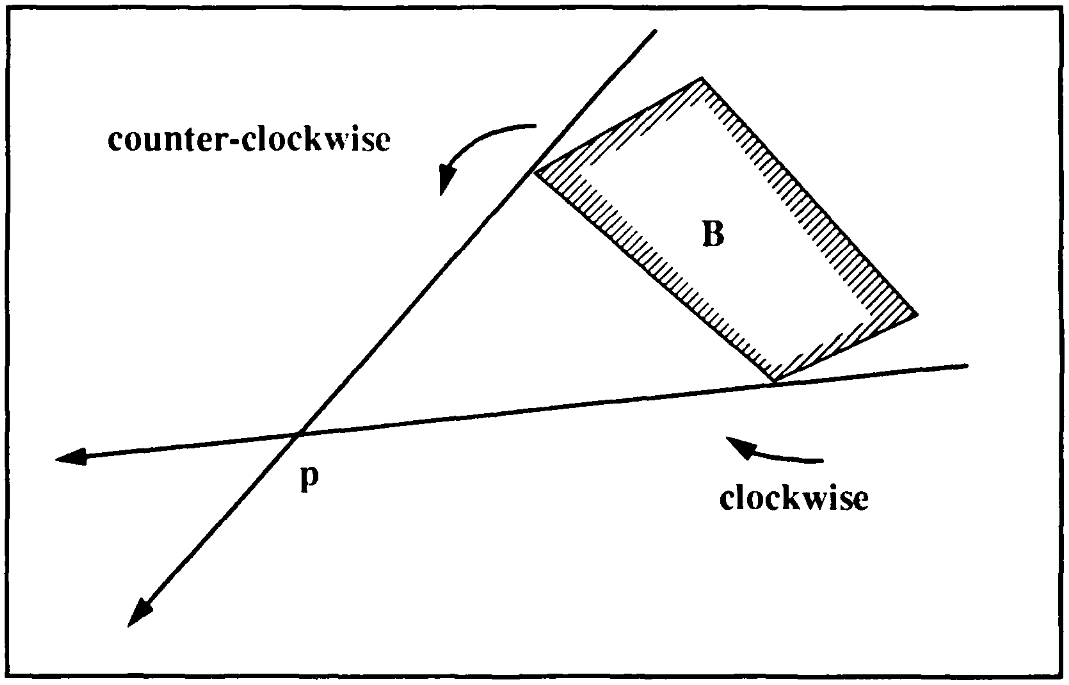


Figure 21- Tangents from a convex polygon to a point

H. Convex Polygons and Common Tangents

Let B_1 and B_2 be two convex polygons. Let a directed line ℓ pass through and osculating vertex first on B_1 and then on B_2, and call these vertices p and q respectively. ℓ is said to be a common tangent of B_1 and B_2 since ℓ is a tangent to both polygons (Figure 22). Common tangents consist of four distinct modes, *plus-plus*, *plus-minus*, *minus-plus*,and *minus-minus*. Notationally, ++, +-, -+, and - - are used to represent the common tangent modes.

ℓ is called a *plus-plus tangent* of B_1B_2 (Figure 23), written $B_1{}^+B_2{}^+$, if

ccw(p,q,previous(q)) and [ccw(p,q,next(q)) or collinear(p,q,next(q))]
and
[cw(q,p,previous(p) or collinear(q,p,next(p))] and cw(q,p,next(p)). **(34)**

ℓ is called a *plus-minus tangent* of B_1B_2 (Figure 24), written $B_1{}^+B_2{}^-$, if

[cw(p,q,previous(q)) or collinear(p,q,next(q))] and cw(p,q,next(q))
and
[cw(q,p,previous(p) or collinear(q,p,next(p))] and cw(q,p,next(p)). **(35)**

ℓ is called a *minus-plus tangent* of B_1B_2 (Figure 25), written $B_1{}^-B_2{}^+$, if

ccw(p,q,previous(q)) and [ccw(p,q,next(q)) or collinear(p,q,next(q))]
and
ccw(q,p,previous(p) and [ccw(q,p,next(p)) or ccw(q,p,next(p))]. **(36)**

ℓ is called a *minus-minus tangent* of B_1B_2 (Figure 26), written $B_1{}^-B_2{}^-$, if

[cw(p,q,previous(q)) or collinear(p,q,next(q))] and cw(p,q,next(q))
and
ccw(q,p,previous(p) and [ccw(q,p,next(p))] or collinear(q,p,next(p))]. **(37)**

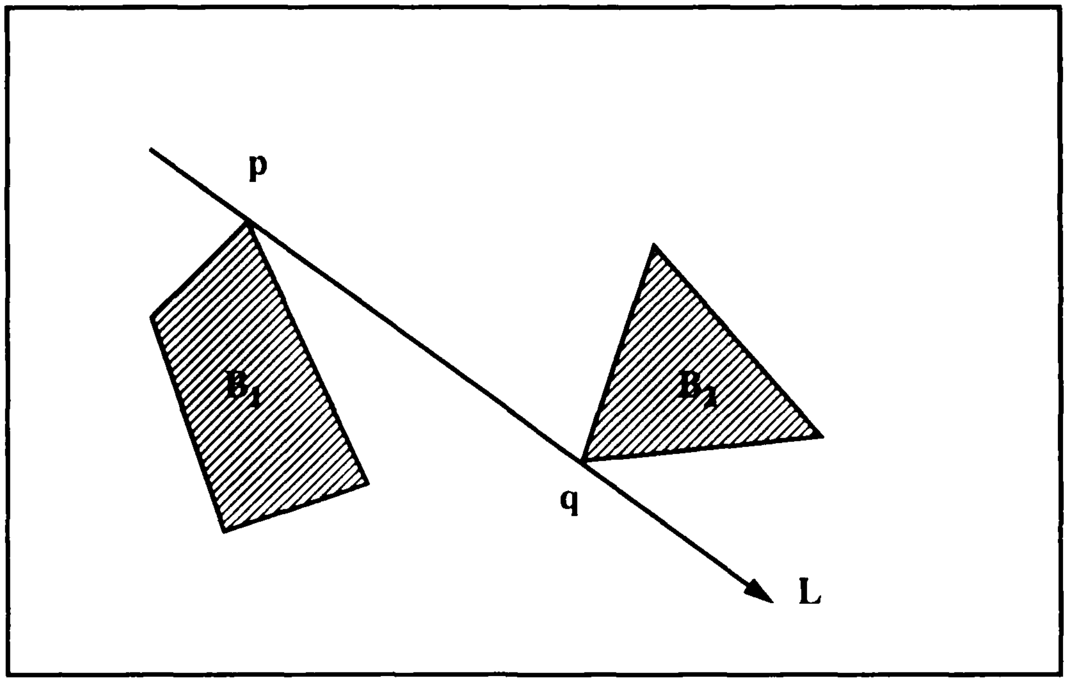


Figure 22 -One Type of Common tangent between two convex polygons

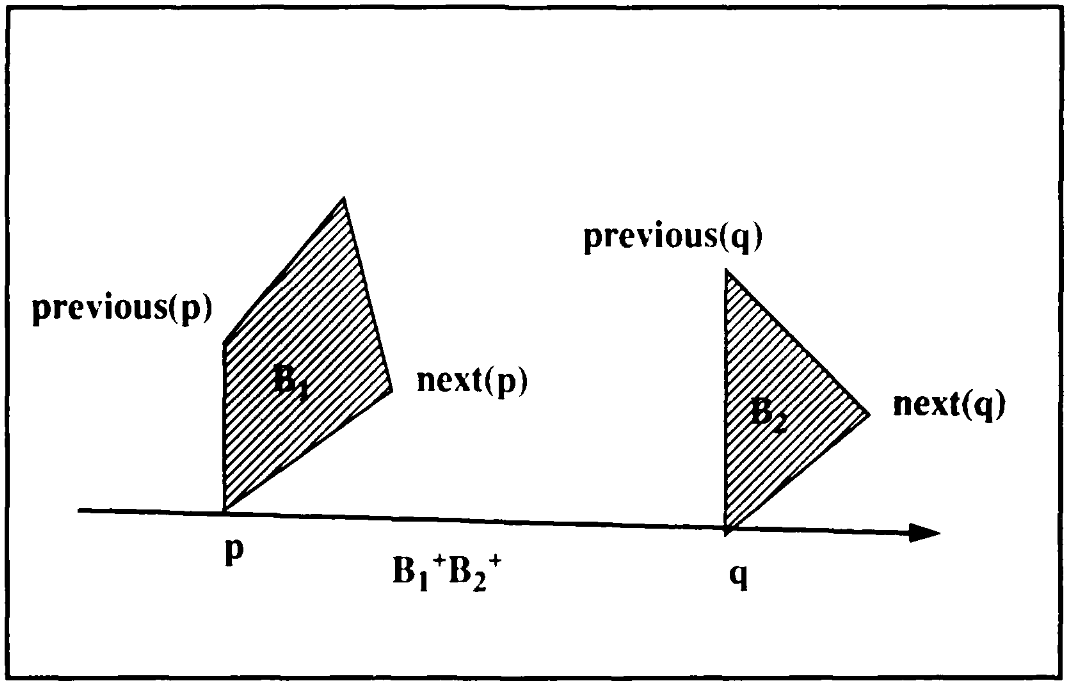


Figure 23 - Plus-plus tangent $B_1^+B_2^+$

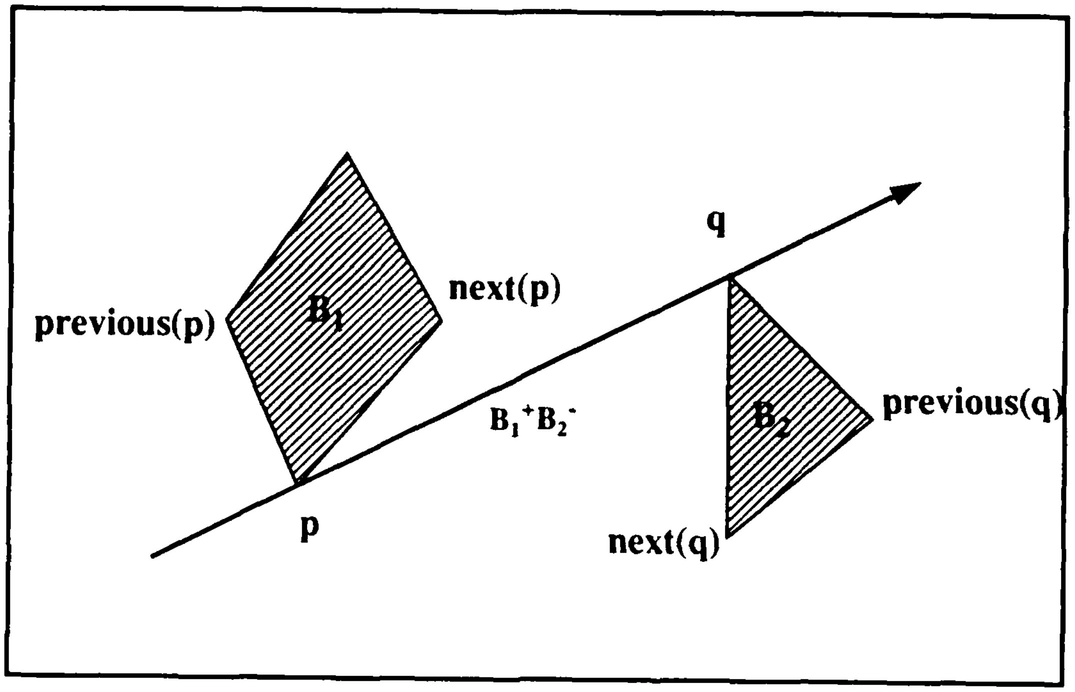


Figure 24 - Plus-Minus Tangent $B_1^+B_2^-$

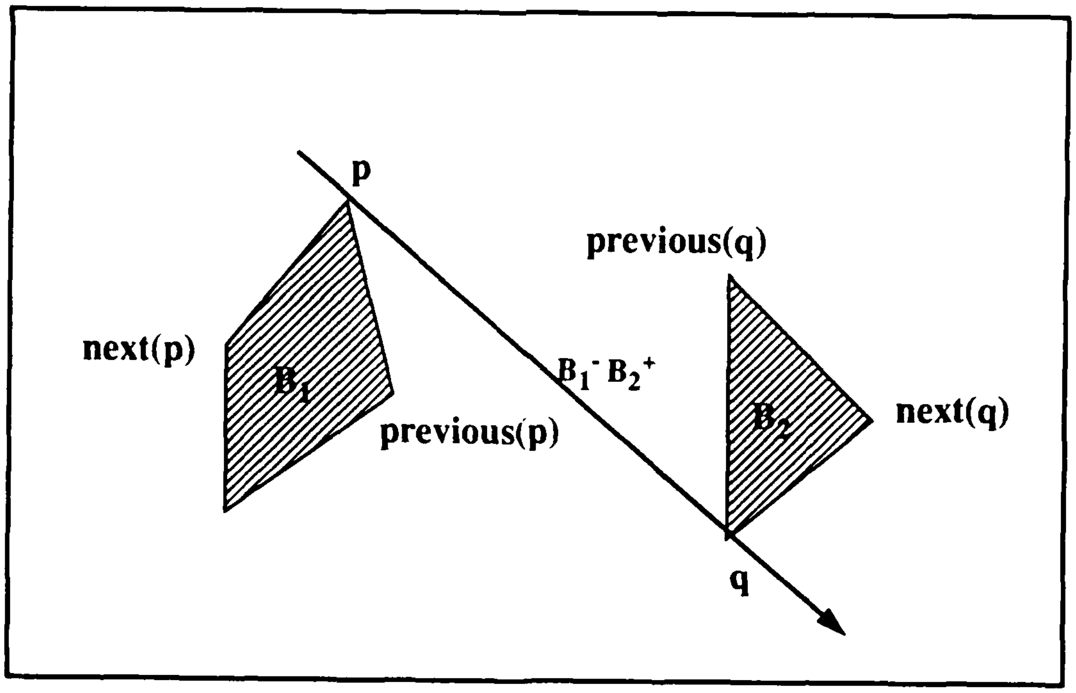


Figure 25 - Minus-Plus tangent $B_1^-B_2^+$

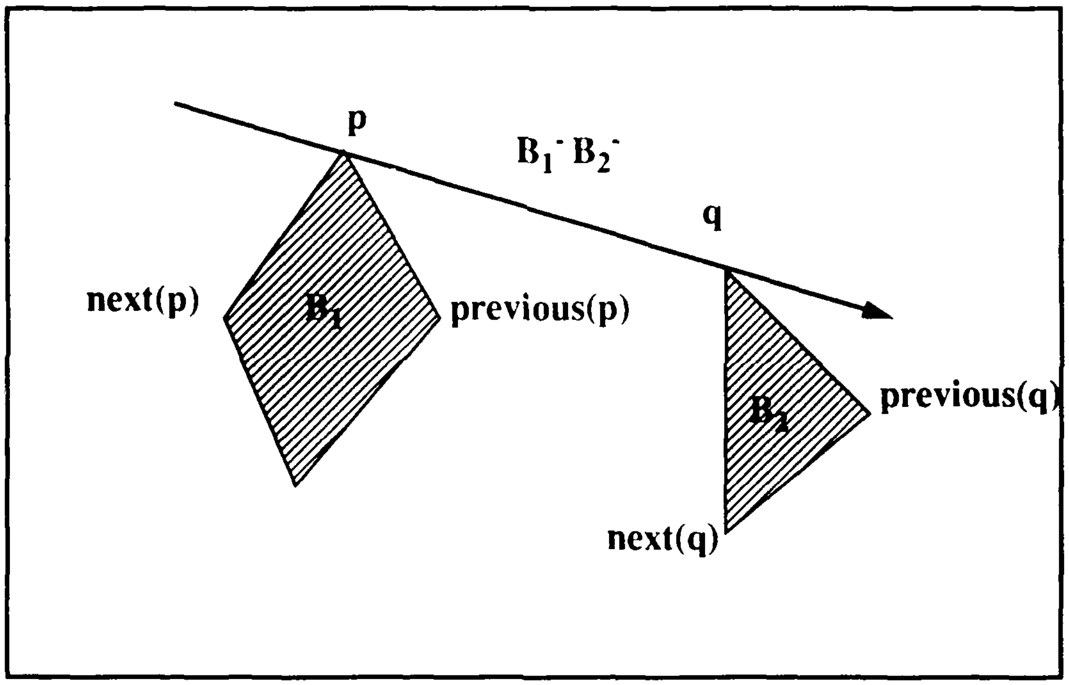


Figure 26 - Minus-Minus Tangent $B_1^- B_2^-$

If two polygons B_1 and B_2 are both convex, there exist exactly one common tangent of each type or mode from B_1 to B_2 and exactly one of each type or mode from B_2 to B_1 (Figure 27). Thus, the existence or validity of a common tangent in one direction means that one also exists in the opposite direction. The tangent modes, however, do not remain the same in all cases. When $\mathcal{L}$ forms the *plus-plus tangent* $B_1^+B_2^+$ there also exists a *minus-minus tangent* $B_2^-B_1^-$ when $\mathcal{L}$'s direction is reversed, and when $\mathcal{L}$ forms the tangent $B_1^-B_2^-$ reversing $\mathcal{L}$'s direction forms $B_2^+B_1^+$. The *plus-minus* and *minus-plus tangents* formed by $\mathcal{L}$, $B_1^+B_2^-$ and $B_1^-B_2^+$, become $B_2^+B_1^-$ and $B_2^-B_1^+$ when $\mathcal{L}$'s direction is reversed (Figure 27).

I. CONVEX SUBPOLYGONS

If either B_1, B_2 or both are concave polygons, there can exist more than one tangent of a given mode between B_1 and B_2, depending on the orientation of the two polygons (Figure 28). It is also possible for a concave polygon to have common tangents onto itself (Figure 29). The fact that more than one tangent of a particular mode may exist when dealing with concave polygons can make reference to a distinct tangent difficult. In order to provide some form of unambiguous symbolical representation of common tangents, a convention for the division and naming of distinct portions of concave polygons is necessary. Tangents can only exist to and from convex vertices of a concave polygon, and the division can be accomplished based on a logical ordering or grouping of these convex vertices. This subdivision will not fully meet the requirement to distinctly reference all possible polygon/tangent possibilities. It does, however, establish a simple and logical convention for construction of and reference to tangents in a world where concave polygons are allowed to exist.

The convention presented here takes any given concave polygon and subdivides it into the smallest number of distinct portions, or *convex subpolygons*. A *convex subpolygon* of

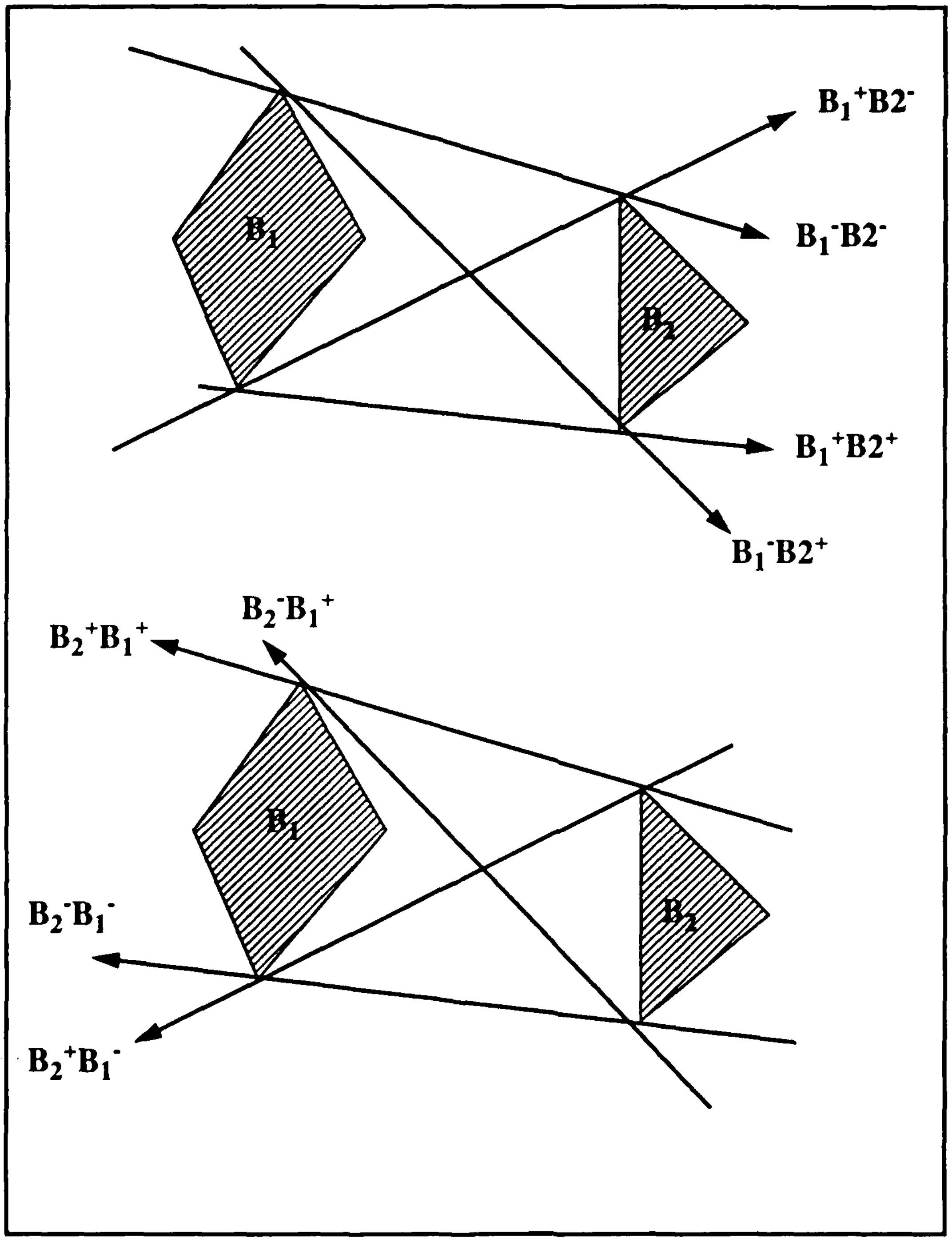


Figure 27 - Eight Modes of Common Tangents Between Two Convex Polygons B_1 and B_2

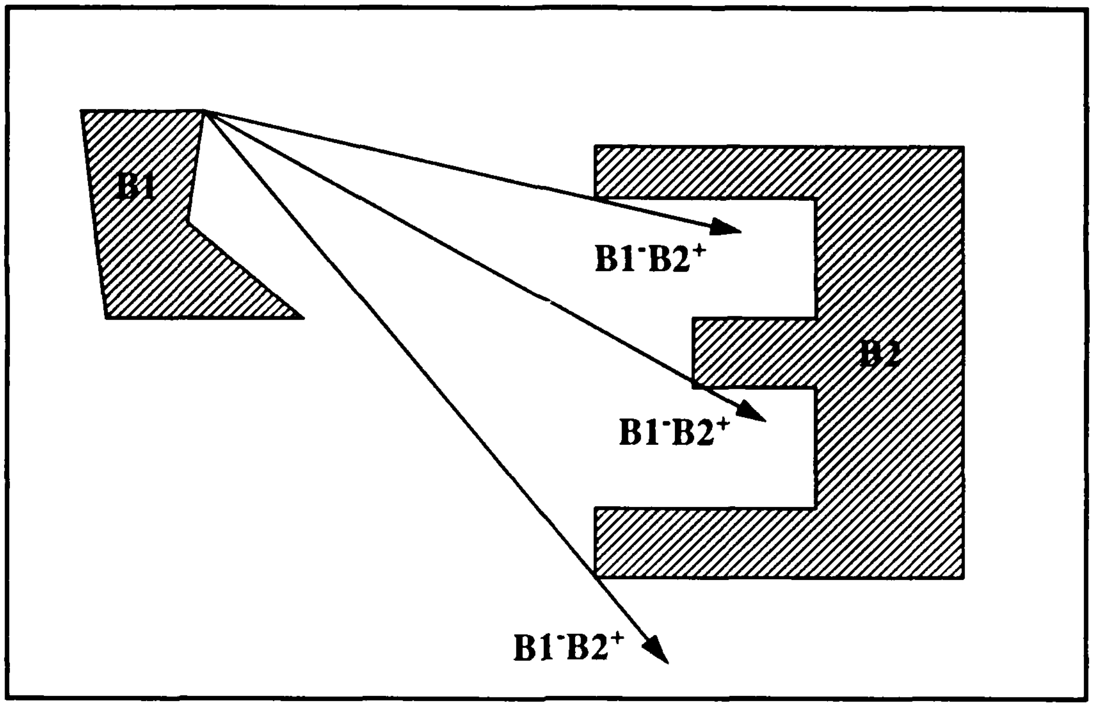


Figure 28 - Multiple Minus-Plus Tangents $B1^-B2^+$

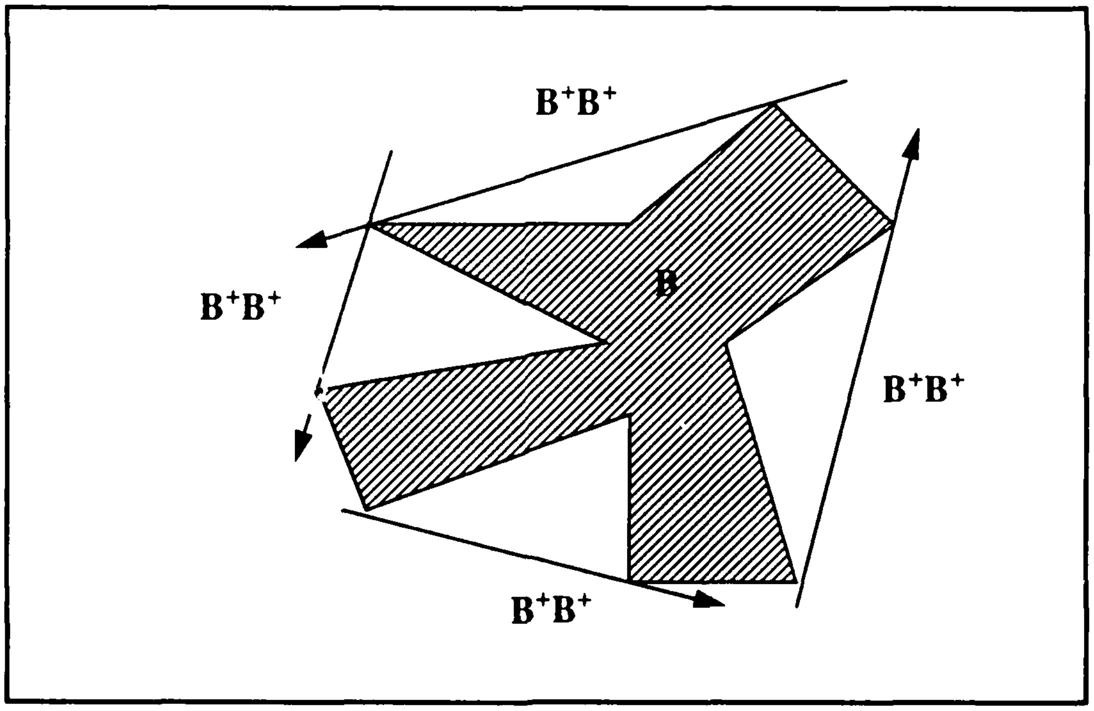


Figure 29 - Multiple Self-Tangents of the Same Mode

a concave polygon is defined as a consecutive group of convex vertices. A convex polygon is the base case where a single *convex subpolygon* contains every vertex in the original vertex set (Figure 30). For a concave polygon, there may be one or more *convex subpolygons*, since each consecutive grouping of convex vertices may be seperated by one or more concave vertices (Figures 31 and 32).

Consider a single concave polygon B represented by the triple

$$\mathbf{B = (}V_B\mathbf{,\ v_0,\ } \boldsymbol{next}\mathbf{),} \tag{38}$$

where V_B represents the vertex set of B such that

$$V_B = \{\, v_0, v_1, \ldots, v_{n-1} \},\tag{39}$$

and n is the number of vertices of B. We wish to divide the vertices of B into two distinct subsets; one consisting of the convex vertices of B, V_B^{convex}, and the other consisting of the concave vertices of B, $V_B^{concave}$. V_B^{convex} is further partitioned into subsets of consecutive or adjacent vertices. These groupings of consecutive convex vertices in V_B^{convex} are labeled B_i, where $0 \leq i \leq q-1$, and q = the number of groupings of consecutive convex vertices in V_B^{convex}. Each of these groupings of vertices outline a distinct portion of the original polygon consisting only of convex vertices, hereafter referred to as a *convex subpolygon*, of the original concave polygon. Every convex vertex of B belongs to one and only one *convex subpolygon*, and

$$V_B^{convex} = V_{B_0} \cup V_{B_1} \cup \ldots \cup V_{B_{q-1}}.\tag{40}$$

The concave polygon B is composed of the *convex subpolygons* $B_0, B_1, \ldots, B_{q-1}$, where each B_i is the *convex subpolygon* corresponding to the ith grouping of consecutive convex

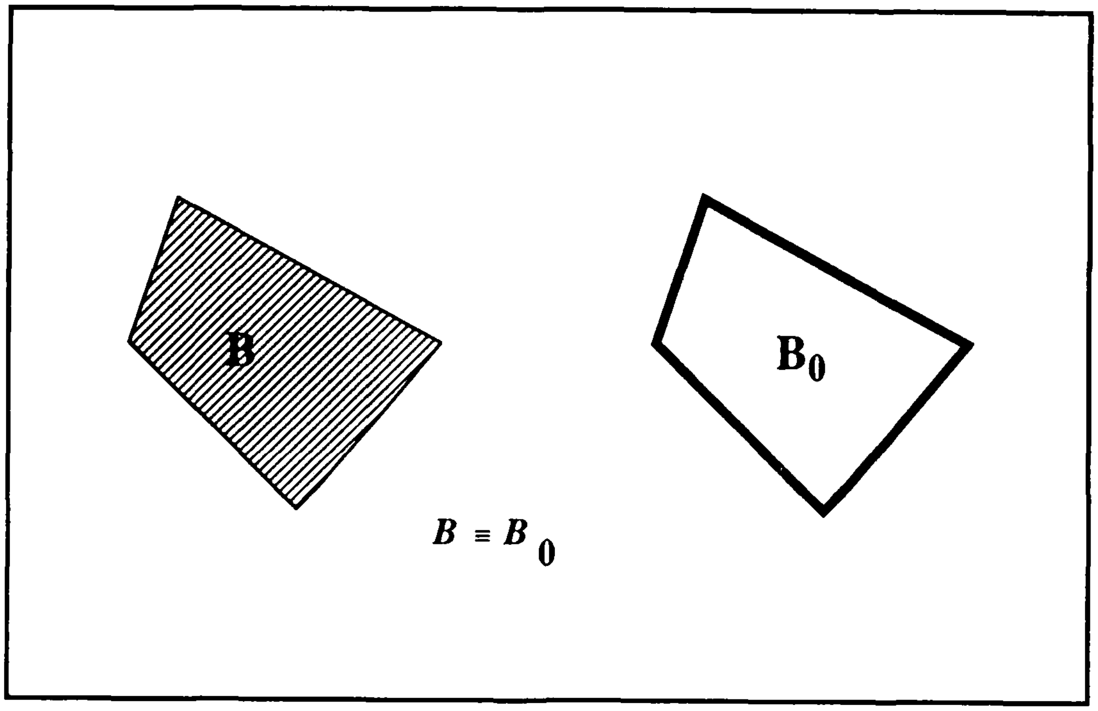


Figure 30 -Equivalence of Convex Polygon and its Convex Subpolygon

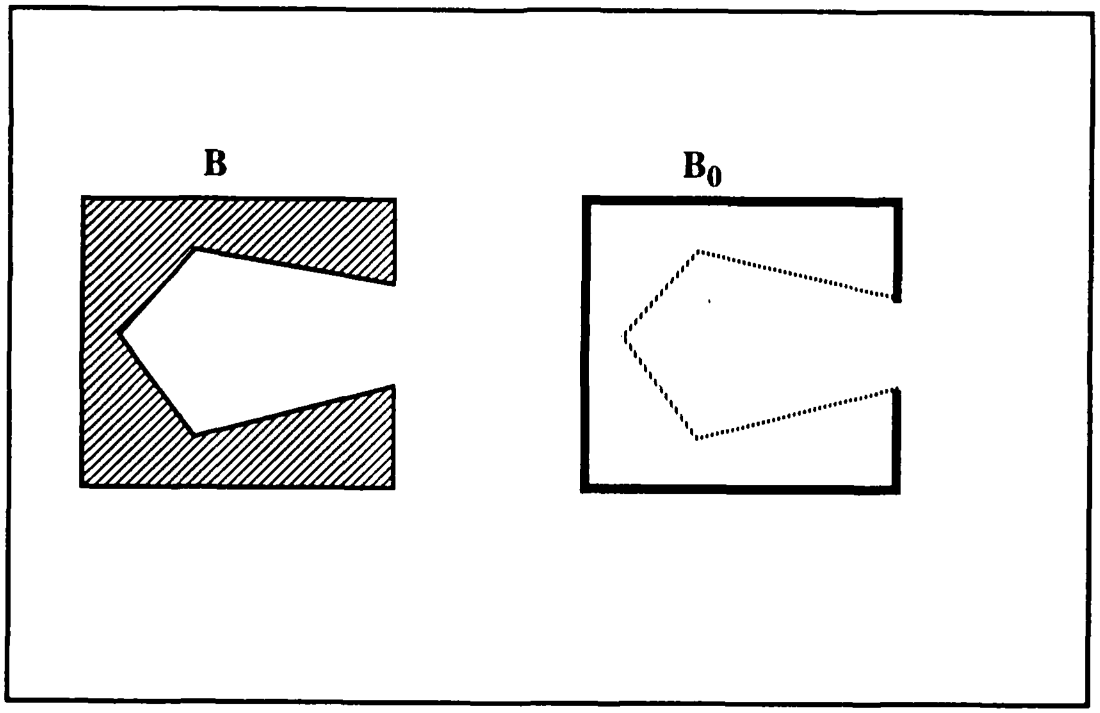


Figure 31 - Concave PolygonRepresented by a Single Convex Subpolygon

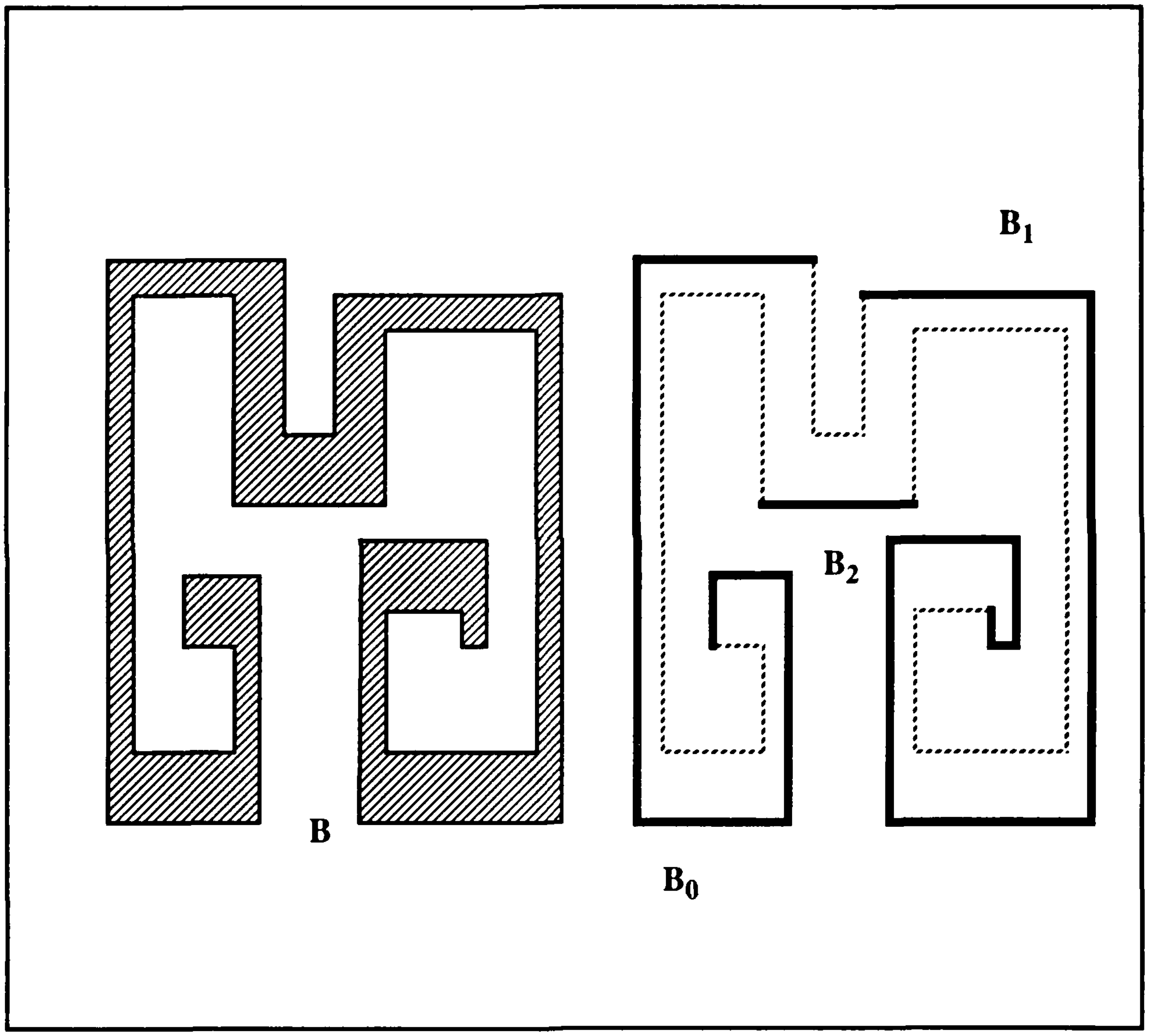


Figure 32 - Single Concave Polygon with Multiple Convex Subpolygons

vertices, plus the set of all concave vertices. Reconsider the polygons and tangents in Figures 28 and 29 after partitioning into convex subpolygons in Figures 33a and 33b.

Procedurally, the vertices of B must be partitioned using only v_0 and the *next* function. This is necessary since no constraints have been imposed concerning the positioning of the vertex v_0 with respect to the convex portions of polygon B except that v_0 is the first vertex used in specifying B. This subdivision of B into consecutive convex subsets is accomplished by locating the first set of consecutive convex vertices beginning after v_0.

Thus, if v_0 is a concave vertex,

$$v_{current}=v_0 \quad \text{and} \quad type(v_{current})=concave, \tag{41}$$

then continue traversing the vertices of B in a counter clockwise direction using the *next* function,

$$v_{current} = next(v_{current}), \tag{42}$$

examining each vertex until one is found which is convex,

$$type(v_{current})=convex. \tag{43}$$

The first grouping of consecutive convex vertices, B_0, is opened and this vertex becomes the first vertex in the *convex subpolygon* B_0. This traversal of the vertices of B in a counter clockwise manner continues, adding each succeeding convex vertex to B_0,

$$type(v_{current})=convex. \tag{44}$$

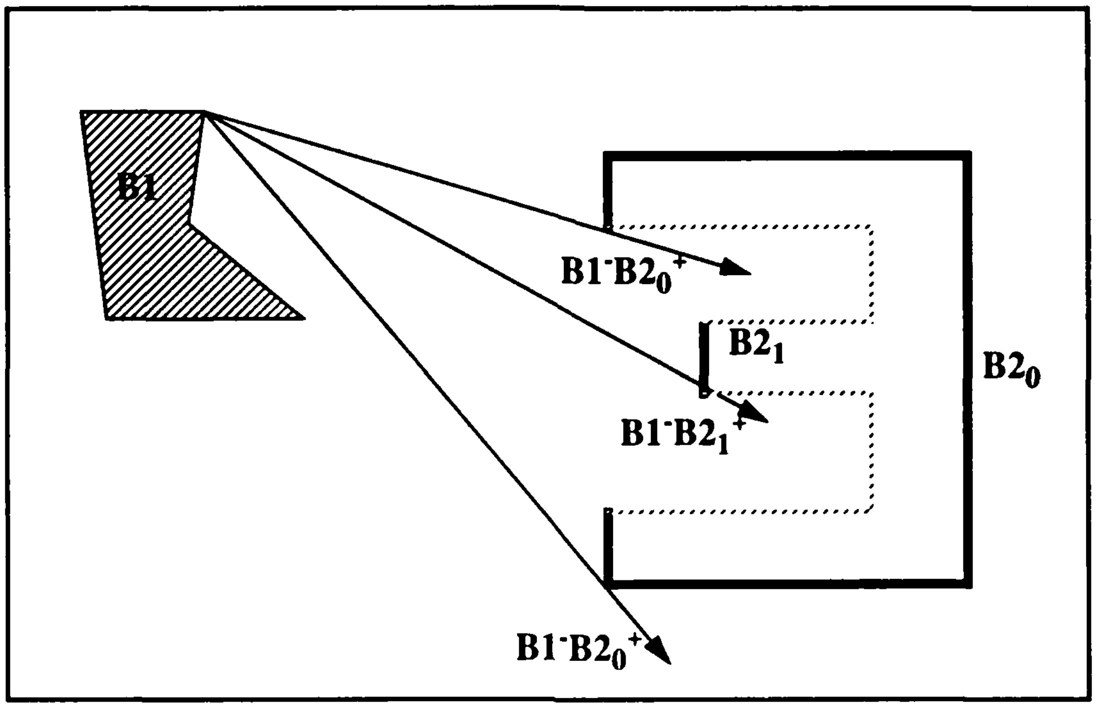


Figure33a - Multiple Minus-Plus Tangent Using Convex Subpolygons

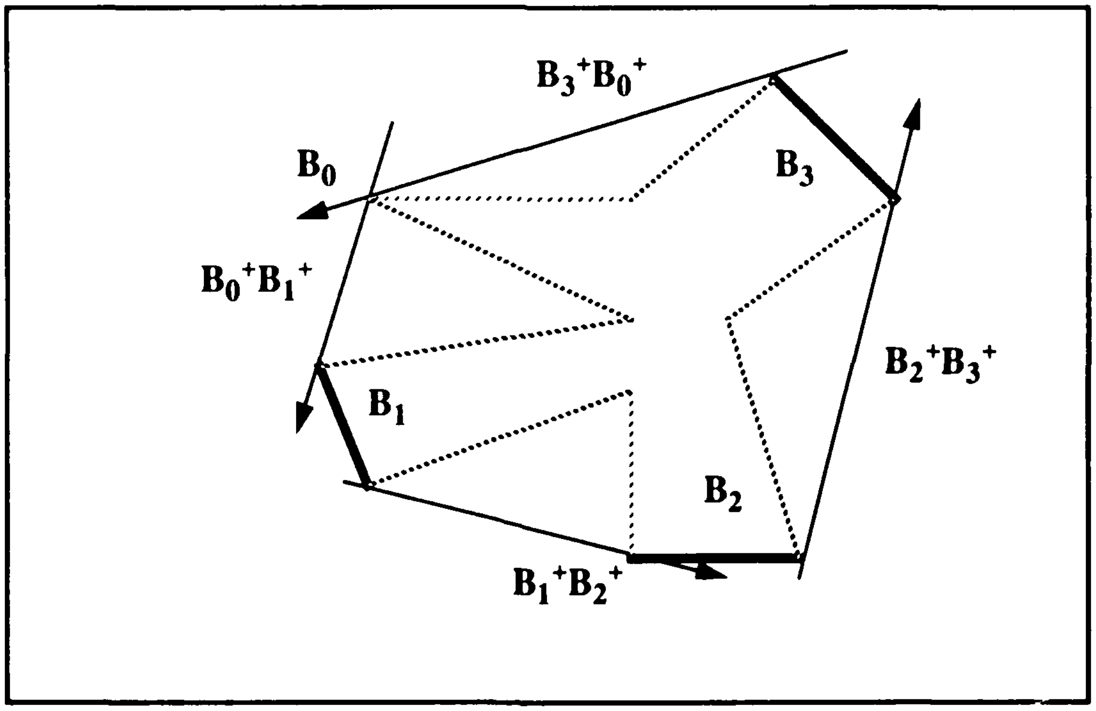


Figure 33b - Multiple Self-Tangents Using Convex Subpolygons

When a vertex is encountered which is concave,

$$type(v_{current})=concave, \tag{45}$$

the *convex subpolygon* B_0 is closed. This counter clockwise traversal continues, creating, opening and closing the successive *convex subpolygons*, $B_0, B_1, \ldots, B_{q-1}$, until all vertices have been examined.

When v_0 is a convex vertex,

$$v_{current}=v_0$$

and

$$type(v_{current})=convex, \tag{46}$$

it is not certain whether this is the beginning, middle or ending element in one of the convex vertex subsets. It is necessary to traverse counter clockwise past v_0,

$$v_{current} = next(v_0), \tag{47}$$

continuing counter clockwise,

$$v_{current} = next(v_{current}) \tag{48}$$

and examining each vertex until one is found which is concave,

$$type(v_{current})=concave. \tag{49}$$

At this point, the processing continues as if v_0 had been a concave vertex, outlined above.

J. COMMON TANGENTS AND CONVEX SUBPOLYGONS

The conditions for locating common tangents for concave polygons follow exactly those for convex polygons, except that like their more complicated geometrical shapes, we must consider a more complicated set of relationships. First, as discussed earlier, there can exist tangents from one portion of a concave polygon to another portion of that same polygon, called self-tangents. Second, the naming convention for distinct portions of concave polygons, or *convex subpolygons*, does not provide a wholly unambiuous tangent mode descriptor. The following section addresses the complexities of concave polygons and common tangents.

Consider a concave polygon B, consisting of only four vertices, and a directed line ℓ existing in B's free space (Figure 34a).

$$B = \{v_1, v_2, v_3, v_4\}. \tag{50}$$

If B is sub-divided into *convex subpolygons* as we have prescribed for a concave polygon, we have

$$B_0 = \{v_1, v_2, v_4\}, \tag{51}$$

shown in Figure 34b. The line ℓ forms a plus-plus tangent, written B^+B^+, using the same criteria as a *plus-plus* tangent between two seperate convex polygons. Reversing the line ℓ's direction would form the *minus-minus* tangent, B^-B^-, just as for two convex polygons. This most simplistic case can be extended to cover all concave polygons which, when partitioned as described, have a single subset of convex vertices, all adjacent to each other and all contained in the convex hull of a concave polygon.

When B's shape becomes more complex, there exist conditions where multiple tangents of any given type can be formed between the same or different *convex*

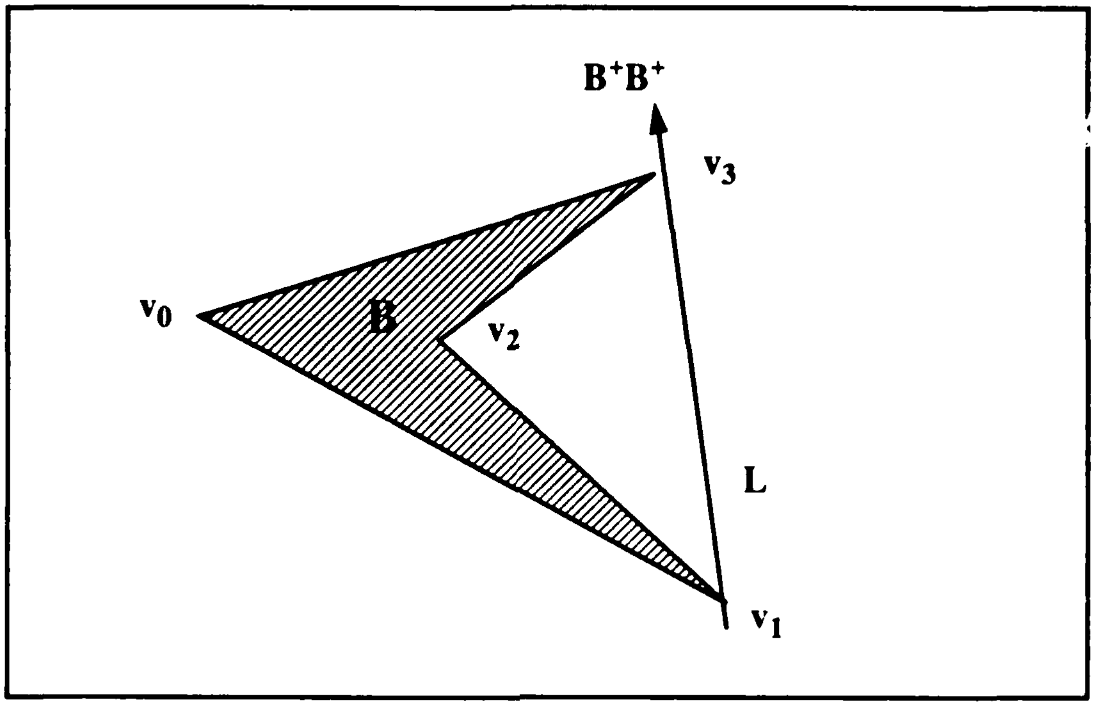


Figure 34a - Concave Polygon and a Single Tangent Line L

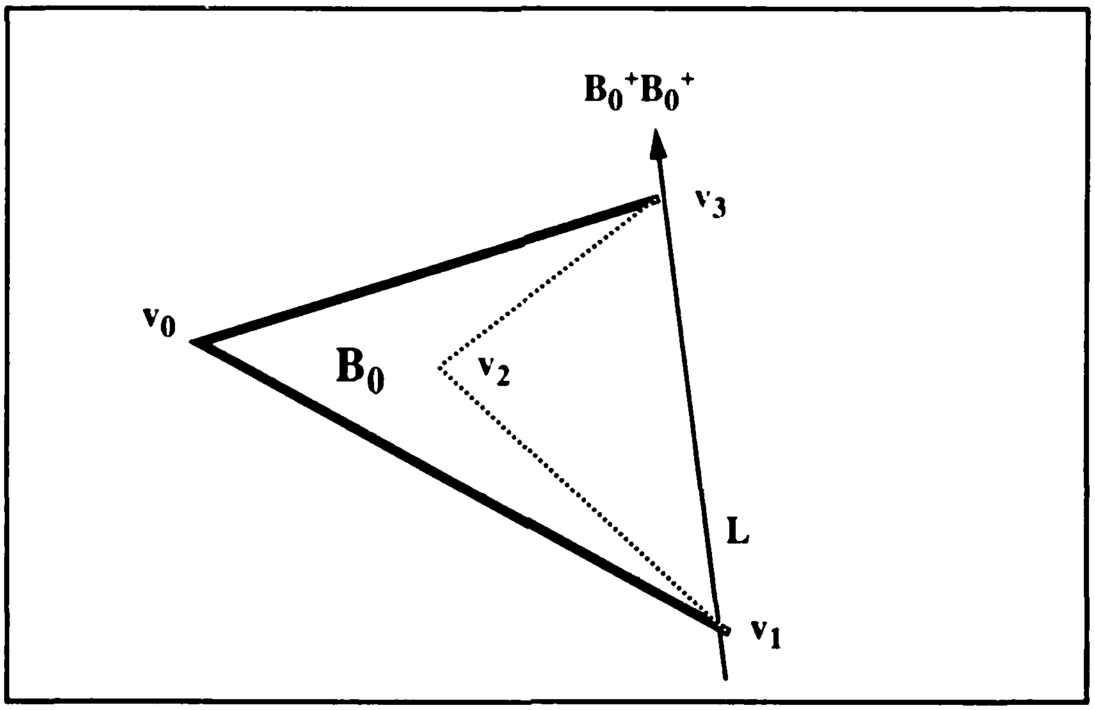


Figure 34b -Convex Subpolygon of Figure 33 with Tangent Line L

subpolygons of a single concave polygon (Figure 35). Partitioning of a concave polygon into convex subpolygons as we have described eases much of the confusion in tangent referencing, clearly ambiguities continue to exist (Figure 36).

Consider another concave polygon B,

$$B = \{v_0, v_1, \ldots, v_{11}\}, \tag{52}$$

partitioned into a single convex subpolygon, B_0,

$$B_0 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_{10}, v_{11}\}, \tag{53}$$

and a point p exsting in B's free space (Figure 37). When p is positioned as shown, it is possible to construct four tangents from p to B, two plus tangents, each labeled B^+, and two minus tangents, each labeled B^-, (Figure 38). The partitioning of B into the convex subpolygon B_0 does not aid in uniquely identifying one plus or minus tangent from another since there still exist two plus tangents, B_0^+, and two minus tangents, B_0^-.

Now consider the pair of minus tangents from p to B_0. If the minus tangent which osculates on v_5 is extended infinitely it would intersect the convex subpolygon B_0 along the edge formed by v_1 and v_2. Extension of the minus tangent line which osculates on v_3 does not result in an intersection with any edge of the convex subpolygon B_0 (Figure 39). Examination of the two plus tangents to B_0 yield the same results. We can now classify the four tangents to the convex subpolygon into two catagories when the tangent lines are extended; those that intersect the convex subpolygon and those that do not. A plus or minus tangent line osculating on a vertex of a convex subpolygon which, when extended infinitely, intersects one of the edges of that convex subpolygon, is said to be an *interior* tangent, as this tangent line provides access to the interior of this convex subpolygon. Similarly, a tangent line which does not intersect one of the edges of its convex

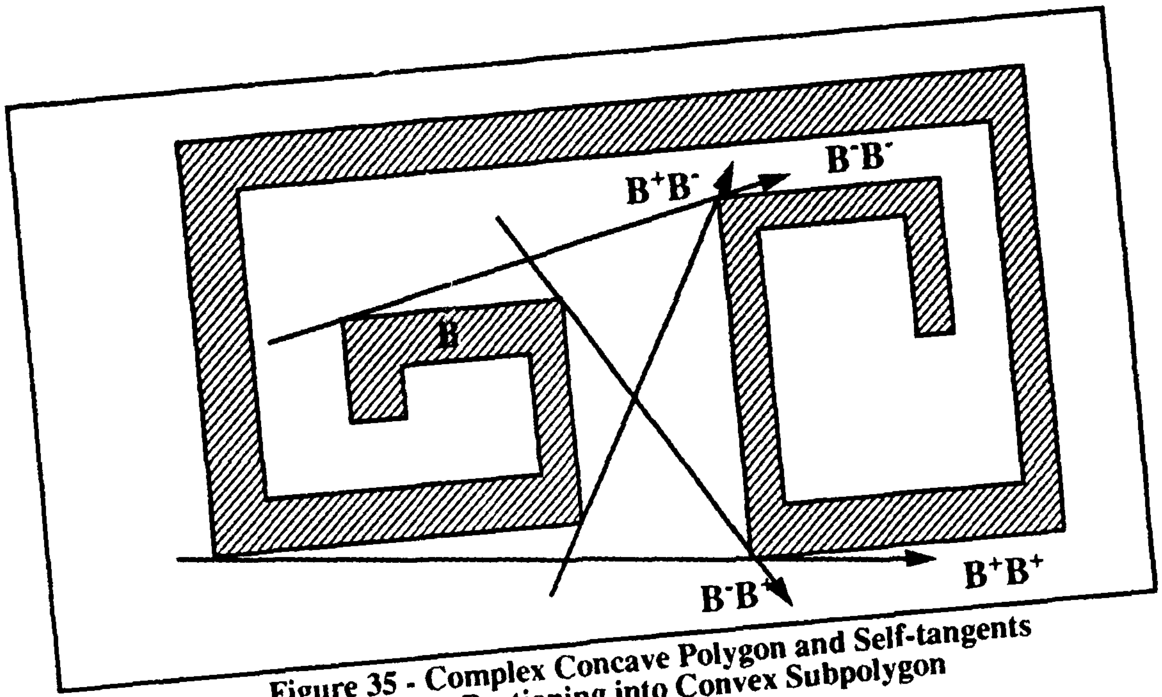


Figure 35 - Complex Concave Polygon and Self-tangents Before Partioning into Convex Subpolygon

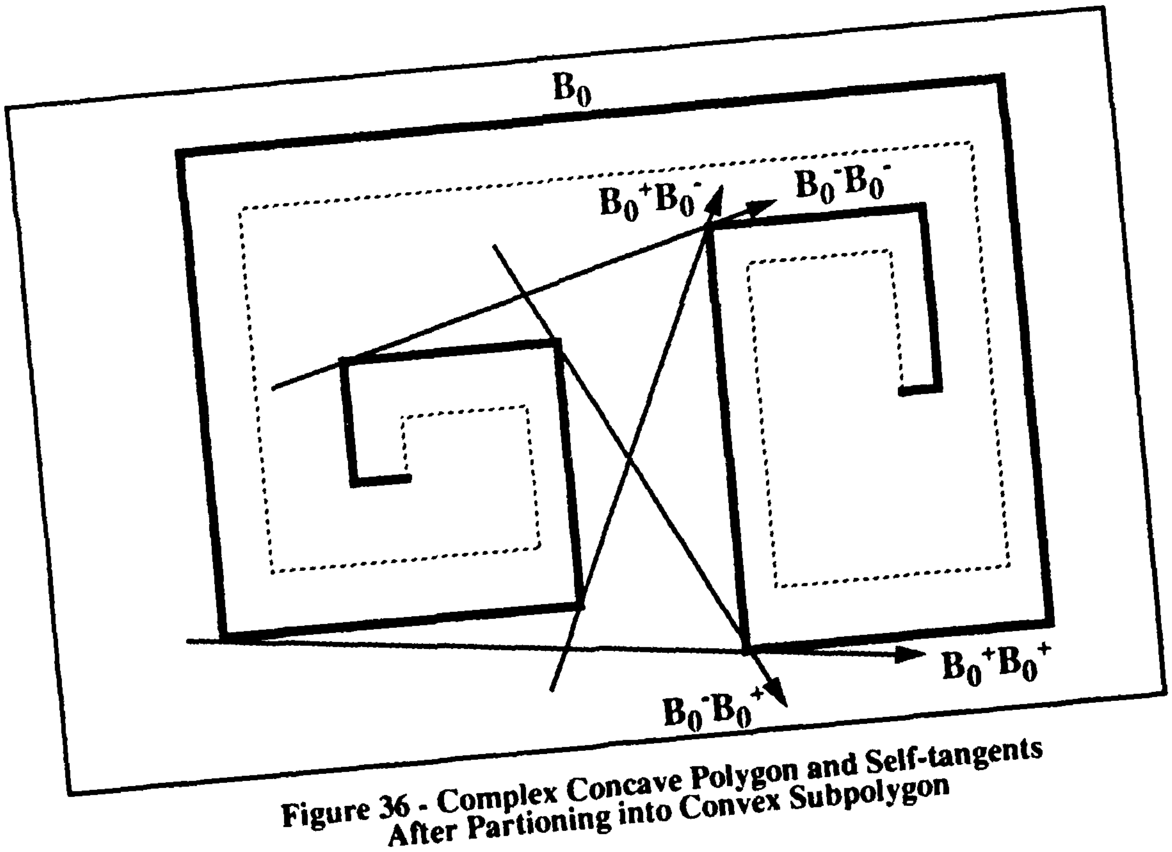


Figure 36 - Complex Concave Polygon and Self-tangents After Partioning into Convex Subpolygon

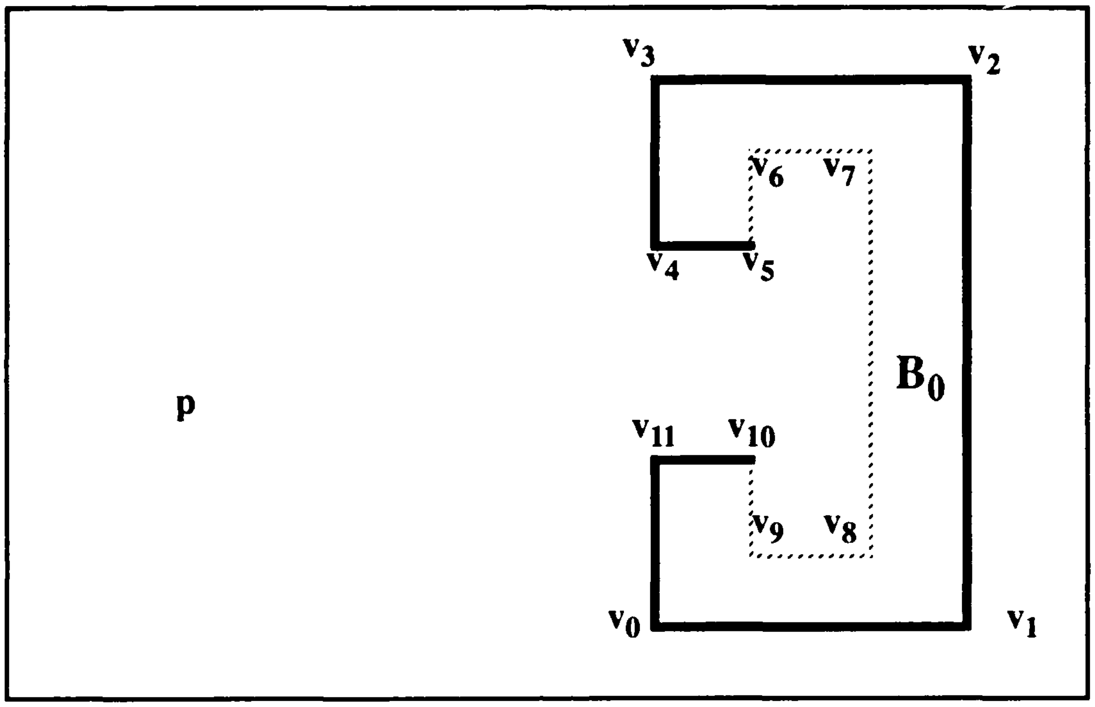


Figure 37 - Partitioning of the Vertices of B into the Convex Subpolygon B_0

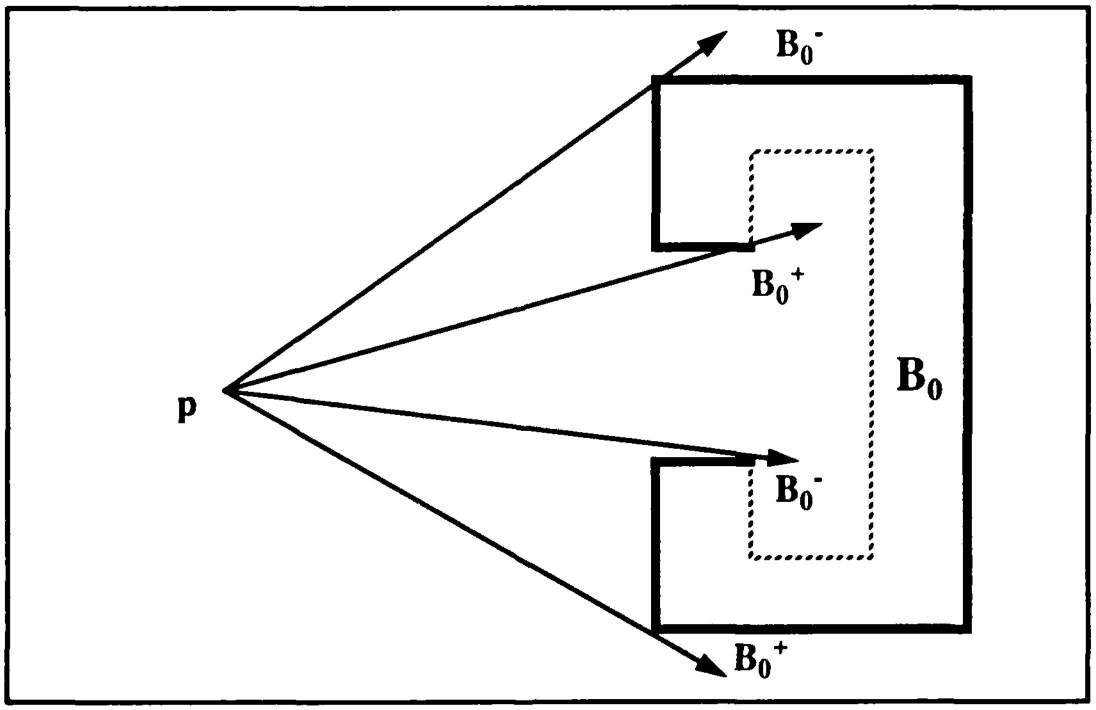


Figure 38 -Multiple Tangents from p to the Convex Subpolygon B_0

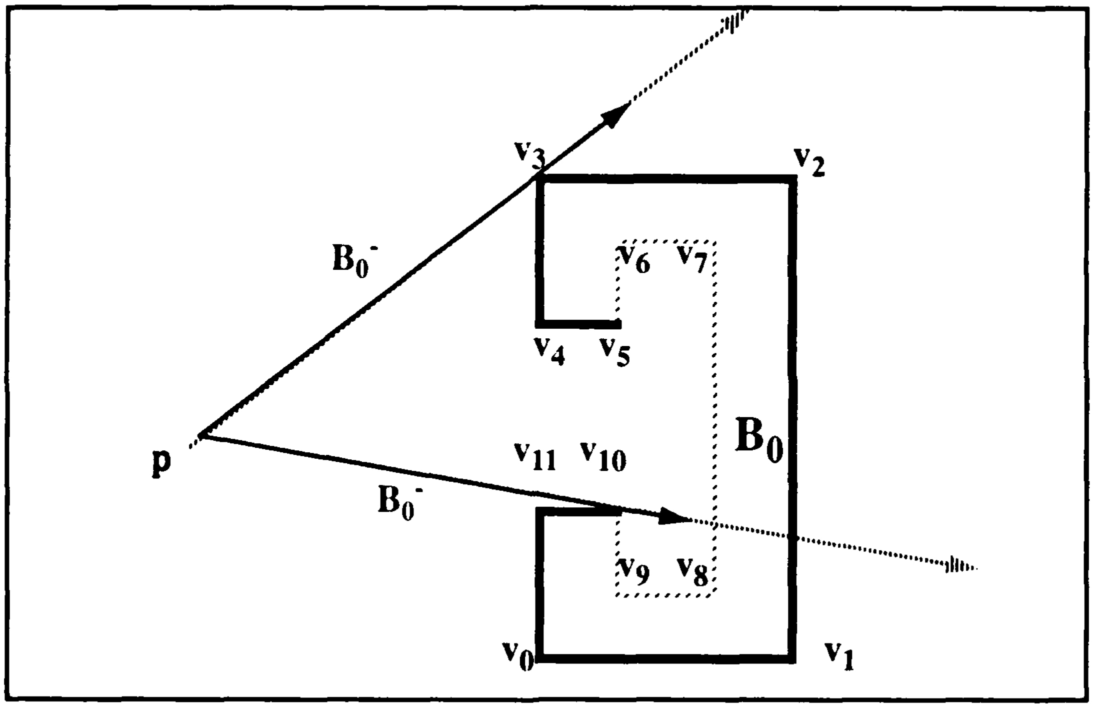


Figure 39 - Extension of the Minus Tangents to the Convex Subpolygon B_0

subpolygon is said to be an *exterior* tangent. Notationally, an exterior tangent is noted using a single tangent mode superscript, in this case B_0^+ and B_0^-, while a double superscript is used for interior tangents as in B_0^{++} and B_0^{--}.

We now have a convention for concave polygons which allows reference to be made to distinct portions of the polygon using convex subpolygons. Additionally, we can not only notationally distinguish between interior and exterior tangents to a convex subpolygon, but can use a simple test for intersection previously oultined to make this determination. What may not be readily apparent is that this approach to concave polygons using convex subpolygons is wholly consistent with the propositions presented earlier concerning tangents to convex polygons. The only sigificant difference is that a convex polygon has only external tangents, while a convex subpolygon has both internal and external tangents.

III. SHORTEST PATHS USING COMMON TANGENTS

A. WORLD DEFINITION AND VISIBILITY

A world, W, consists of a set of polygons

$$W = \{B_1, B_2, \ldots, B_m\},$$

where the space divided by the polygon boundaries are consistent. Consistency implies that the intersection of the filled space of all polygons in W is empty,

$$\textit{filled-space}(B_1) \text{ intersect } \textit{filled-space}(B_1), \ldots, \textit{filled-space}(B_1) = \phi.$$

The free space of W, called F, is defined as the union of the free space of the set of polygons in W . F includes the boundaries of the polygons in W, allowing paths to touch the polygons, but not enter into the filled space of any polygon. Two points, p_1 and p_2, existing in W's free space or F, are said to be visible if the segment $\overline{p_1p_2}$ does not cross the boundaries of the polygons in W, written *visible*(p_1,p_2). An osculating polygon, or one which has one or two adjacent vertices on $\overline{p_1p_2}$, is not considered to be a crossing (Figures 40 and 41), consistent with our definition of F. A visibility test for a segment, especially for a tangent, is a frequent operation in spatial reasoning. These tests are easily constructed using the crossing tests for two line segments outlined in Chapter II.E. of this thesis.

Generally, we must determine whether a line segment L intersects any of the edges of the polygons which partition a world into free space and filled space. This is accomplished by sequentially testing the line L for crossing with the edges of all of the polygons in the world in question.

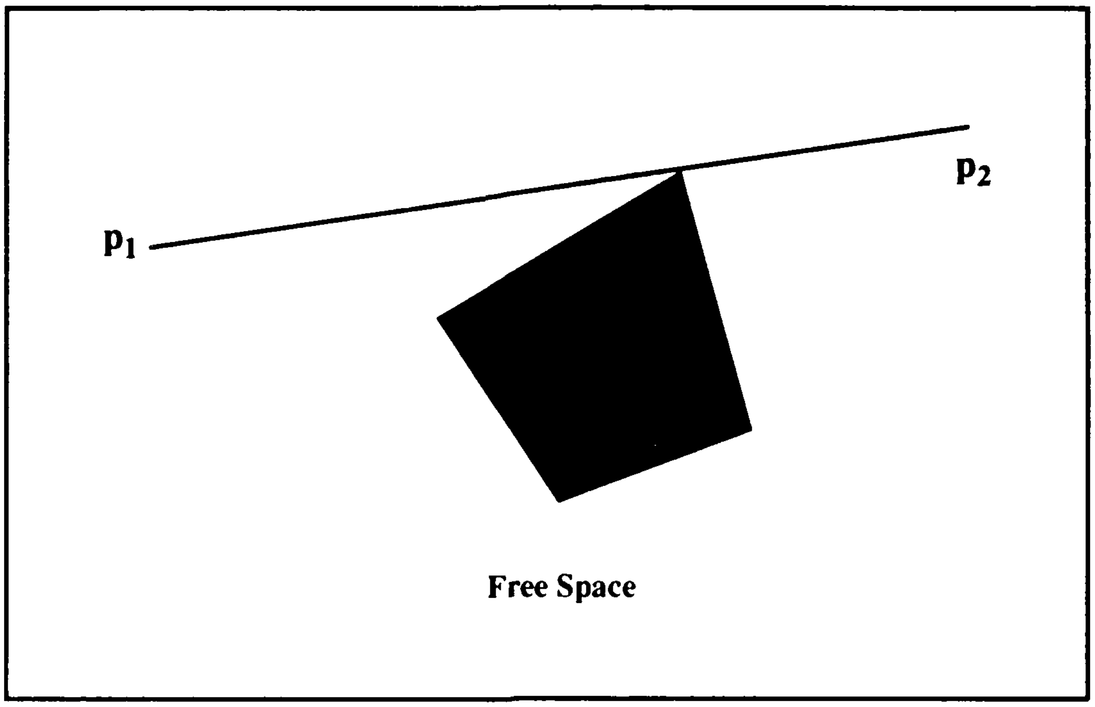


Figure 40 -Segment $\overline{p1p2}$ Osculating on a Single Vertex of a Polygon

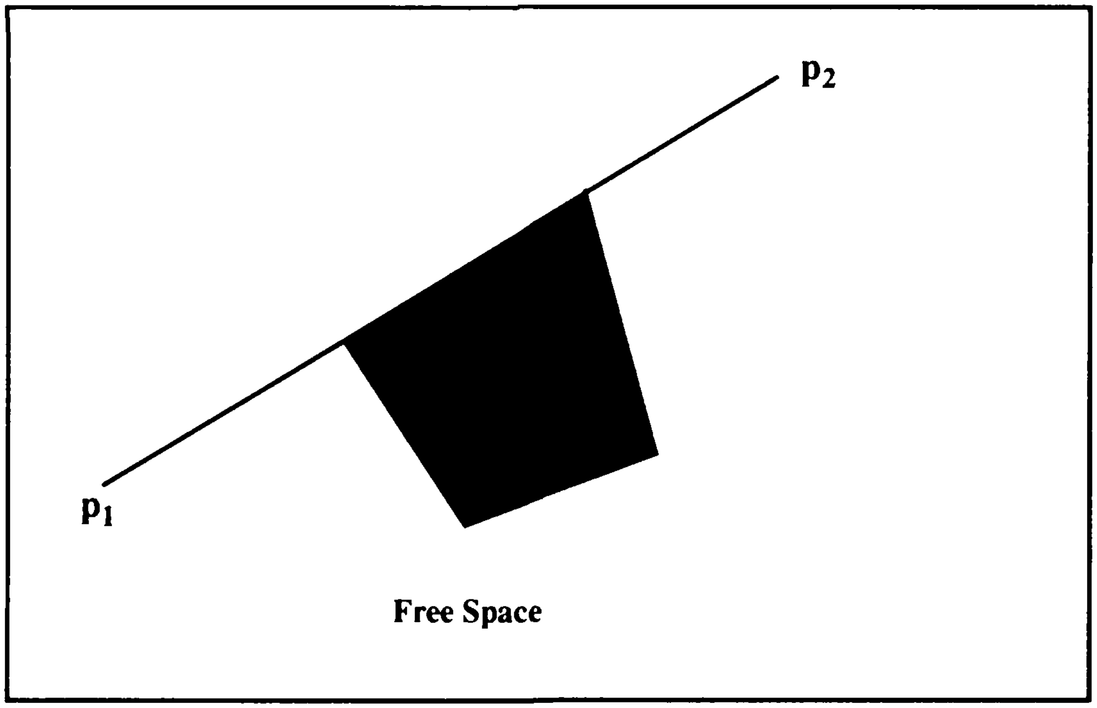


Figure 41 -Segment $\overline{p1p2}$ Osculating on aTwo Vertices of a Polygon

Consider a world W such that

$$\mathbf{W=\{ B_0, B_1,\ldots,B_m\}} \tag{54}$$

where each polygon in W is specified as

$$\mathbf{B_i = (V_{Bi}, v_0, \mathit{next})} \tag{55}$$

and $0 \leq i \leq m\text{-}1$, and a line segment L, given by its two end points, p_1 and p_2. We begin by selecting B_0 as the current polygon, and v_0 of B_0 as the current vertex,

$$\mathbf{polygon_{current}=B_0} \tag{56}$$

and

$$\mathbf{v_{current} = v_0.} \tag{57}$$

The edge formed by $v_{current}$ and $next(v_{current})$ is tested for crossing with the line L using the crossing test previously introduced.

$$\begin{gathered}[\mathit{order}(p_1, p_2, v_{current}) \neq \mathit{order}(p_1, p_2, \mathit{next}(v_{current}))\,] \\ \textbf{and} \\ [\mathit{order}(v_{current}, \mathit{next}(v_{current}), p_1) \neq \mathit{order}(v_{current}, \mathit{next}(v_{current}), p_2)].\end{gathered} \tag{58}$$

When true, the test indicates that the line segment L crosses the edge formed by $v_{current}$ and $next(v_{current})$ and the testing is discontinued. Otherwise, it is necessary to test the next edge of the polygon for a crossing with L by selecting the next vertex as $v_{current}$,

$$\mathbf{v_{current} = next(v_{current})} \tag{59}$$

and testing the edge formed by $v_{current}$ and *next*($v_{current}$). The crossing test of the edges continues until crossing has resulted and the test is terminated, or the last vertex of $polygon_{current}$ is reached,

$$\mathbf{next}(v_{current}) = v_0. \tag{60}$$

When the latter is the case, the next polygon in W becomes the current polygon and the testing of polygon edges is repeated as outlined above. It is important to note that for a polygon with *n* vertices, *n-1* crossing tests are necessary to confirm that the line segment does not cross one of the edges of the polygon. Determining that the line segment *L* is visible in a world with *m* polygons and *n* vertices requires *n-m* crossing tests.

B. PATH EQUIVALENCE CLASSES

In the polygonal world we have described, there exist an infinite number of paths which can be constructed between a selected start point, ***S***, and goal point, ***G***. We are interested in finding the optimal path in terms of distance traveled through this world to connect *S* and *G*. The boundaries of the polygons which define the world are considered as part of the free space, ***F***, where paths are permitted to exist, while paths are prohibited in filled space. *F* is a connected, non-empty closed subset of the plane *R*x*R*, where *R* is the set of real numbers.

We define a path π in *F* as a continuous function such that

$$\pi = (\pi_x, \pi_y) : I \rightarrow F \tag{61}$$

where I is the unit closed interval [0, 1]. A point anywhere on the continuous path p is given by

$$\pi(s) = (\pi_x(s), \pi_y(s)) \tag{62}$$

with s element of $[0, 1]$. The end points of π are given by $\pi(0)$ and $\pi(1)$, the start and goal of π respectively. Since the number of paths existing in F sharing the same end points, $\pi(0)$ and $\pi(1)$, is infinite, finding the optimal path from S to G is will be intractable without a means to reduce the number of possibilities to consider.

Consider a world W consisting of two convex polygons B_1 and B_2 and the paths π_1 through π_5 , all sharing common end points (Figure 42). Examination of the five paths visually indicates that π_1 and π_2 are similar, as are π_3 and π_4. However, π_5 is not similar to any of the other paths. Relying on the definition of an equivalence relation from the field of algebraic topology [Hilton 1969], we can formalize this intuitive understanding of the differences and similarities between these paths.

Two paths π and π' in F are "equivalent" if they share common end points and we can continuously transform π into π' without crossing the boundaries of the polygons in W and keeping the end points fixed. The analogy can be made to that of two loose strings which are anchored at the same end points. If we can deform one string through stretching or shrinking until it is exactly the same length and follows exactly the same path as the other, then we have performed such a path transformation. Formally, the equivalence of π and π' can be written as

$$\pi \cong_F \pi' \tag{63}$$

if and only if there exists a continuous function $\eta : I^2 \rightarrow F$ and such that

$$\begin{gathered} \eta(0,s) = \pi(s) \\ \textbf{and} \\ \eta(1,s) = \pi'(s) \end{gathered} \tag{64}$$

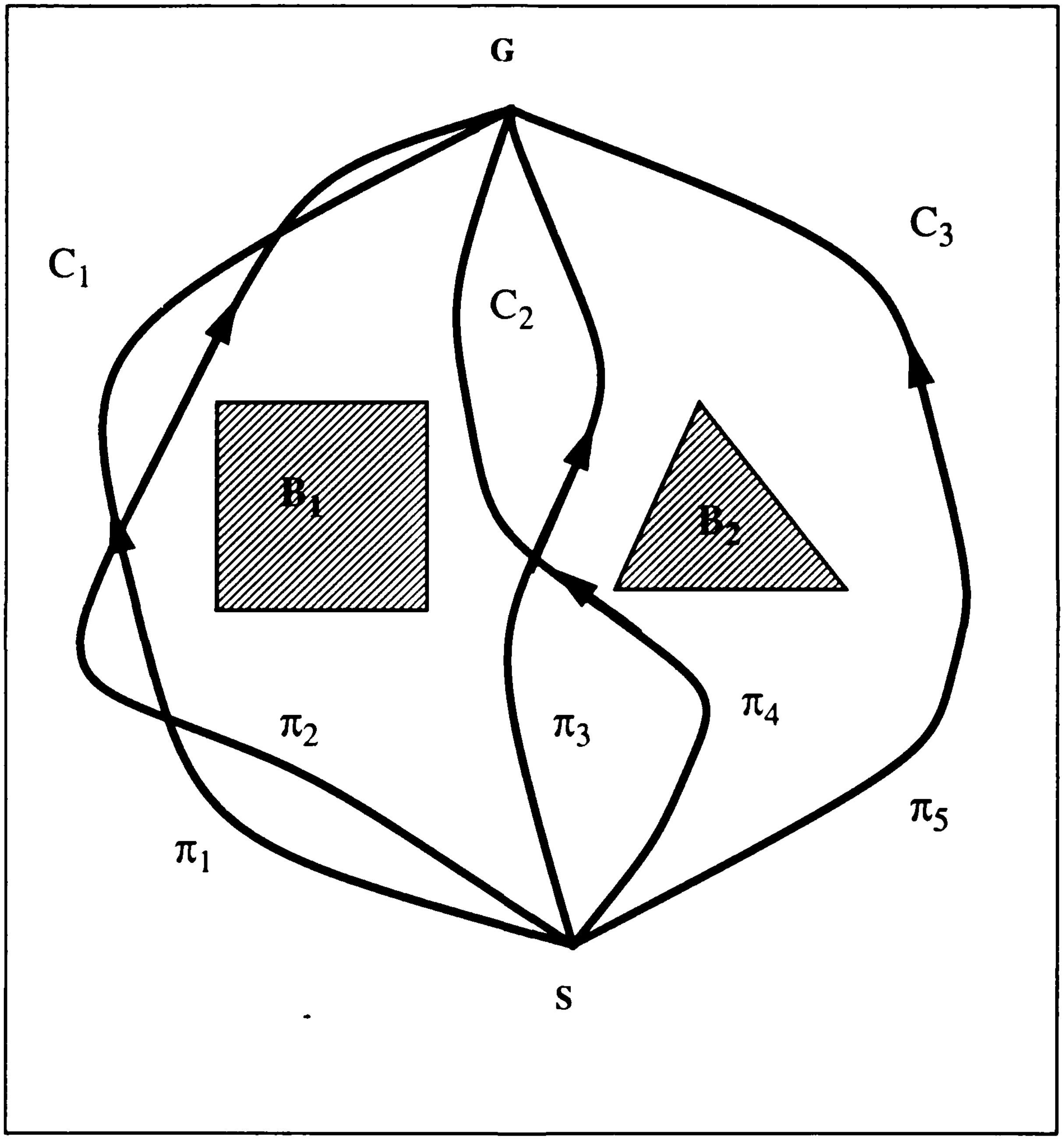


Figure 42 - Five Sample Partial Paths in a Two Polygon World

for all s an element of I, and

$$\eta(t, 0) = \pi(0)$$
and
$$\eta(t, 1) = \pi'(1) \qquad (65)$$

for all t an element of I. When the free space, F, is understood, we can simply write $\pi \cong \pi'$. Likewise, we can state that if

$$\pi \cong \pi' \qquad (66)$$

then

$$\pi(0) = \pi'(0)$$
and
$$\pi(1) = \pi'(1). \qquad (67)$$

When two such paths are equivalent, we can say that $\cong$ establishes an equivalence relation among the various classes of paths which may exist in F. If $\pi \cong \pi'$, these paths traverse the same regions of F and are considered *homotopic* in F, and an equivalence class of paths under the relation $\cong$ is said to be a *homotopy* class in F. We can now formally restate our intuitive observations concerning the paths previously considered in Figure 42 using the path equivalence relationships. The paths π_1 and π_2, which appear similar, belong to the same *homotopy* class C_1 since $\pi_1 \cong \pi_2$, and π_3 and π_4 are in the same *homotopy* class C_2 since $\pi_3 \cong \pi_4$. The path π_5 has no other paths in its homotopy class, C_3, since there does not exist a continuous function to transform any of the paths in F without violating the boundaries of either B_1 or B_2.

Consider another world W with four paths existing in its free space as shown in Figure 43. Clearly these paths all share common end points, however, paths π_2 and π_3 cross over themselves while π_1 does not. We can observe that π_2 has two points along its path, s_1 and s_2, where $\pi(s_1) = \pi(s_2)$ with $0 \leq s_1 < s_1 \leq 1$. In this situation, the path π_2 is said to have a *loop*. The path π_3 also has a loop. When a path does not cross itself, as is the case for paths π_1 and π_4, the path is said to be *loop-free*.

When a path equivalence class contains a loop-free path, the entire path equivalence class is called loop-free. Path π_2 is an example of a path that has a loop, but is equivalent to the loop-free path π_1 since we can transform this looped path into an equivalent one without a loop. The path π_3, however, is not equivalent to any loop-free paths since we cannot transform π_3 into a loop-free path without violating the boundaries of the polygon B. Thus, π_3 belongs to a path equivalence class which is not loop-free.

Note that it is possible to construct an infinite number of homotopy path classes with loops even in a simple world with only one polygon by simply adding another turn around the polygon to each subsequent path. A path which makes k counter-clockwise rotations around a polygon and another which makes $k+1$ counter-clockwise rotations clearly belong to distinct homotopy path classes. This set of paths with loops is infinite, while the set of loop-free paths consist of a finite number of loop-free homotopy classes. It is much more attractive in the optimal path searching process to deal with this finite set of path classes without loops than the infinite set of paths which are not loop-free.

C. TANGENT SEQUENCES AND SHORTEST PATHS

Consider a world W consisting of a finite number of convex polygons. The set T = T(W) of tangent modes of this world is defined as

$$\mathbf{T(W) \equiv \{ B+, B- \mid B \text{ an element of } W \}}$$

A tangent sequence σ over T is a finite sequence of tangent modes such that no subsequence of B^+B^- or B^-B^+ with B an element of W. The empty tangent sequence is denoted by ε. The set of tangent sequences is expressed as T(W)*, where * is an element of {+,-,++,--}, following the conventions of language and automata theory.

Now consider an example world W consisting of two polygons A and B. Allowable tangent modes are A^+, A^-, B^+, and B^-; and possibly A^{++}, A^{--}, B^{++}, and B^{--} if A or B are allowed to be concave polygons. Examples of tangent sequences for W consisting of convex polygons only are

$$\mathbf{T(W)^* = \{ \varepsilon, A^+, A^-, B^+, B^-, A^+B^+, A^+B^-, \ldots, B^-A^+, A^-A^-, \ldots \}.}$$

Note that the sequence A-A- appears as an allowable tangent sequence. This representation signifies a tangent sequence which loops around the polygon A one full rotation (Figure 44). The tangent mode is repeated once for each loop around a polygon, permitting tangent sequences of $A_0^+A_1^+ \ldots A_n^+$ and $B_0^-B_1^- \ldots B_m^-$ to represent *n* full counter-clockwise rotations about A and *m* full clockwise rotations about B.

We can now describe the various paths in this world using a series of tangent sequences, each of which represents the shortest path among a homotopy class of paths. Tangent sequences are an enumeration of zero or more polygon tangent lines and modes, coupled with zero or more polygon edge traversals, which, when interpreted according to the formal definition presented below, define a path from a given start point to a goal point. This path definition method using tangent sequences is another form of representation of path equivalence classes in a polygonal world. As will be shown, tangent sequences provide a method for translating paths with loops into equivalent loop-free paths.

Before formally defining a method for interpretation of tangent sequences, several examples of paths and their tangent sequence equivalents are presented. A simple world W consisting of three convex polygons, A, B and C, can consist of many paths between a fixed start and goal point (Figure 45). It is possible to construct a path directly from ***S*** to ***G*** (Figure 46) without having the path osculate on or intersect any of the polygons. This is the empty tangent sequence, or ε. The tangent sequence $A^-B^+C^+$ (Figure 47a) represents a path which runs from ***S***, then clockwise around A, counter-clockwise around B and C, and then to ***G***. Tangent line traversals are represented as l_i, where $0 \leq i \leq n-1$, and n is the number of tangent lines covered by the path. Edge traversals, which can be empty, are represented as k_j, where $1 \leq j \leq m$, and m is the number of polygons on which the path osculates. Thus, the path $A^-B^+C^+$ can also be described as $l_0k_1l_1k_2l_2k_3l_3$. Similarly, the path $C^+C^+B^+A^-$ (Figure 47b) can be denoted as $l_0k_1l_1k_2l_2k_3l_3$. This introduction to the notation of tangent line and edge traversals embellishes the formal definition of tangent sequence interpretation which follows.

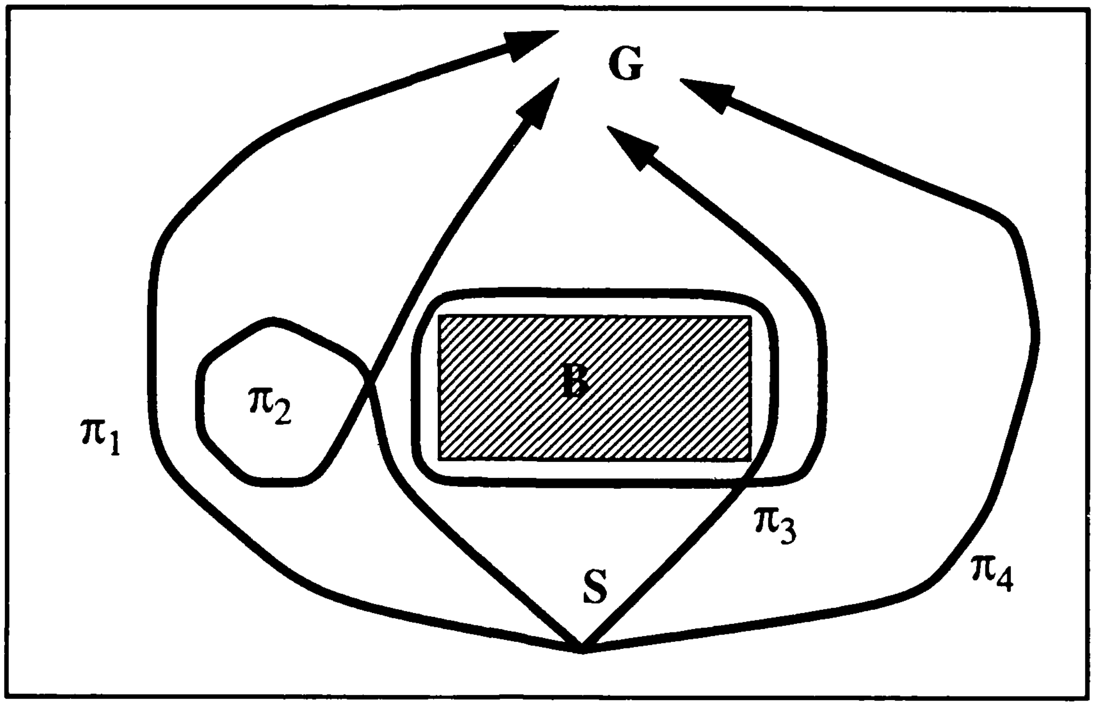


Figure 43 - Four Paths with Common End Points

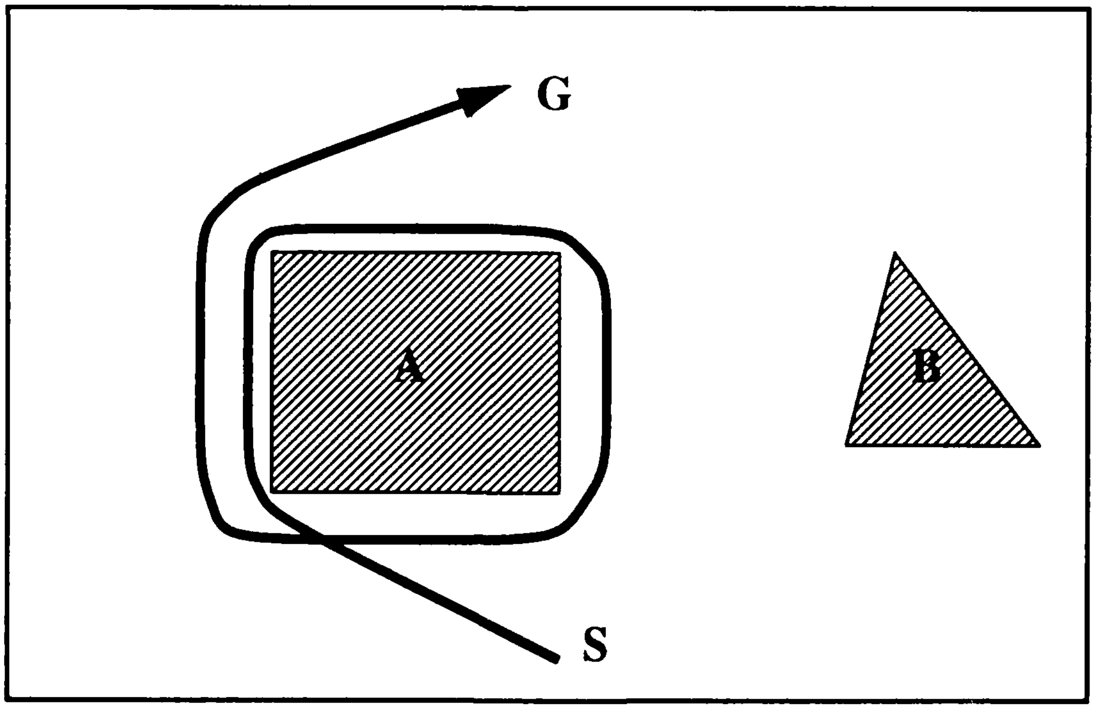


Figure 44 - Tangent Sequence A^-A^-

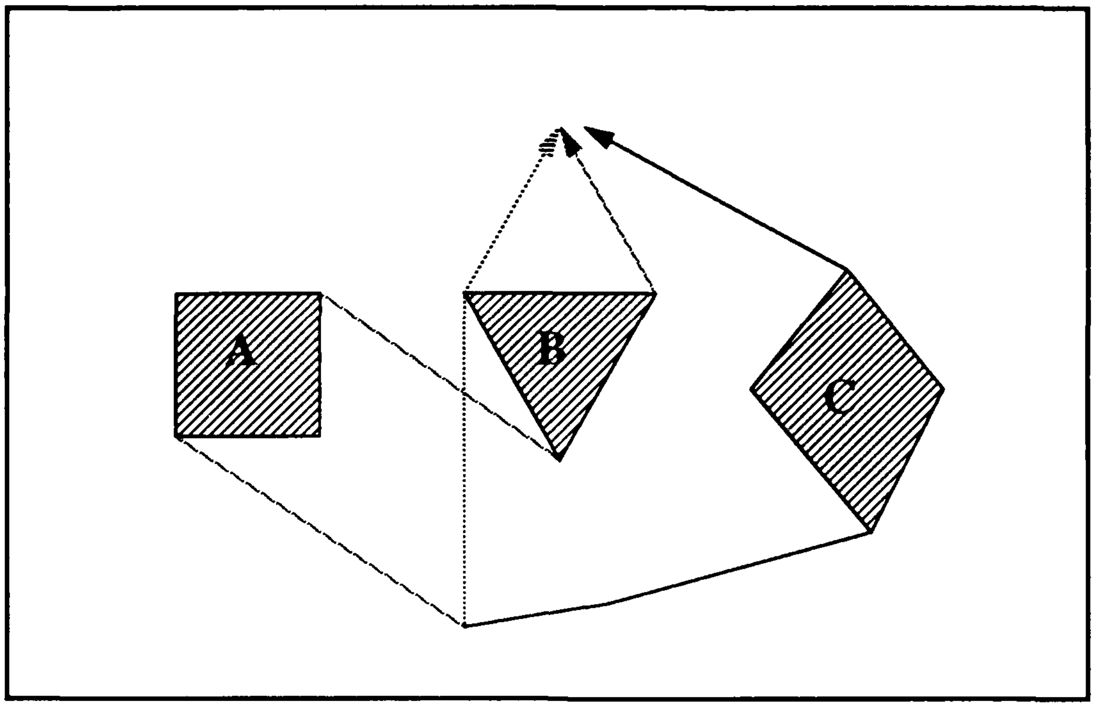


Figure 45 - Possible Paths in a Three Polygon World

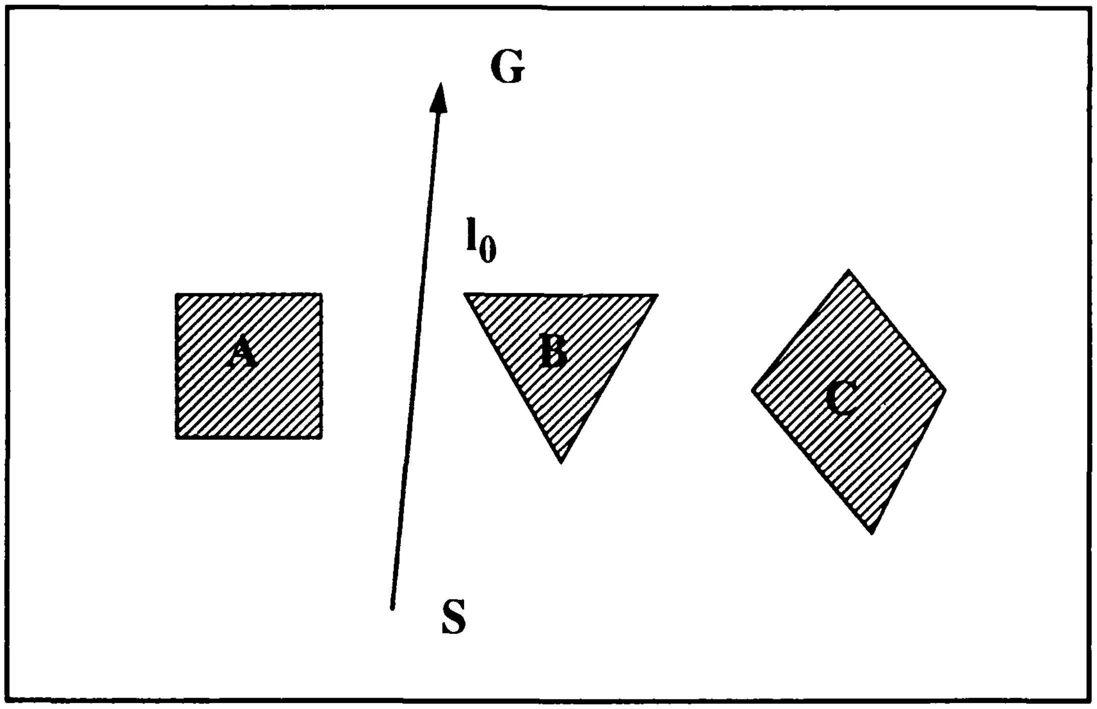


Figure 46 - An Example of an ε Path Where G is Visible from S

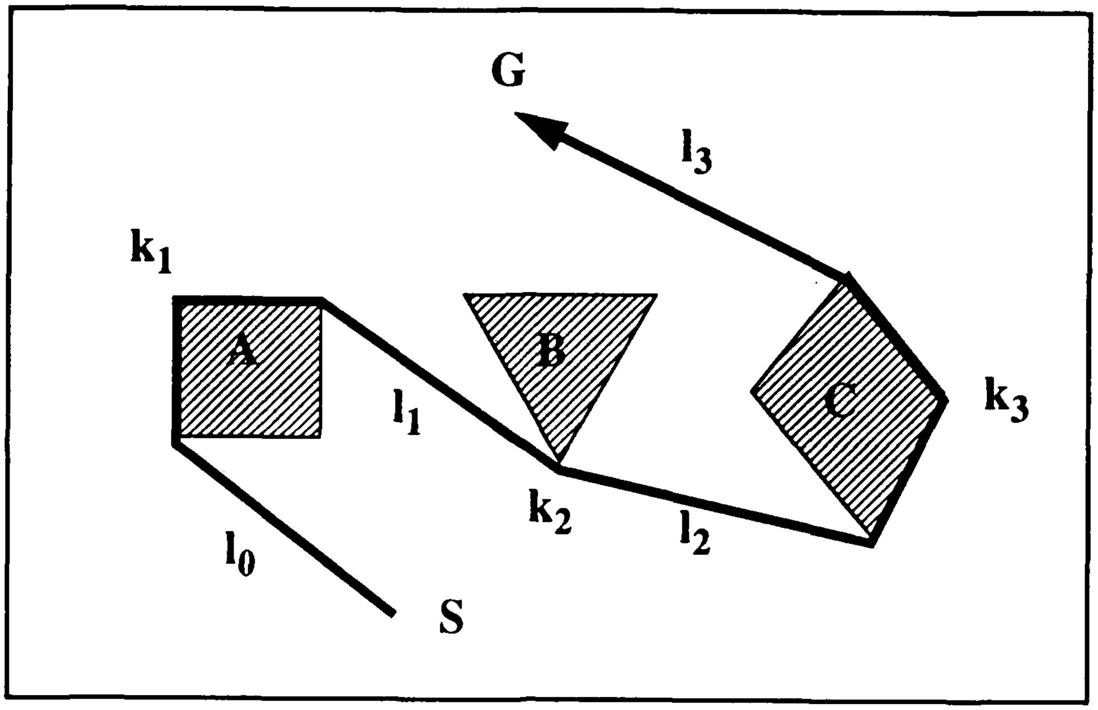


Figure 47a - The Tangent Sequence $A^-B^+C^+$ or $l_0k_1l_1k_2l_2k_3l_3$

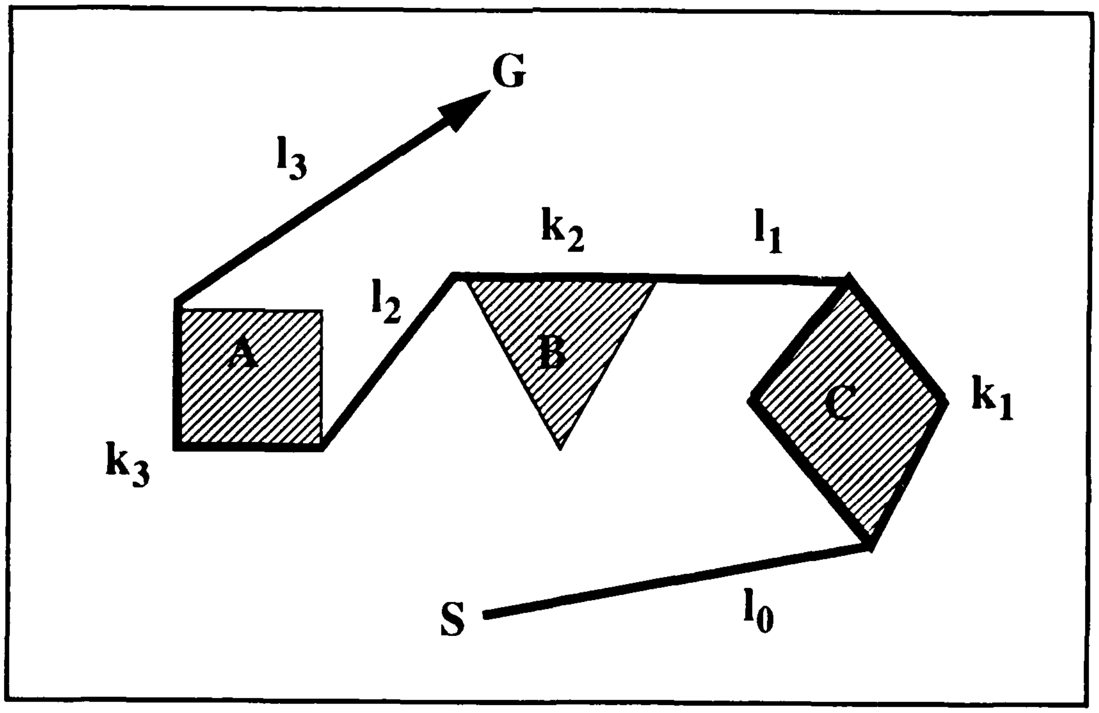


Figure 47b - The Tangent SequenceC$^+$C$^+$B$^+$A$^-$ or $l_0k_1l_1k_2l_2k_3l_3$

A tangent sequence σ element $T(W)^*$ with a start point S and a goal point G is interpreted into a path $\pi(\sigma)$ by the following two rules.

(I) If $\sigma = \varepsilon$,

$$\pi(\sigma) = \overline{SG} \textbf{ if visible(S, G)} \tag{68}$$

where visible(S,G) means S and G are visible in this world. If they are not visible, the value $\pi(\varepsilon)$ is undefined.

(II) If $\sigma = B^{*1}_{i1},\ldots,B^{*q}_{iq}$ where $q \geq 1$, $B_{i1},\ldots,B_{iq}$ are elements of W, and $*1,\ldots,*q$ are elements of $\{+,-,++,--\}$, then $\pi(\sigma) = l_0 k_1 l_1,\ldots,k_q l_q$

where

(i) the right hand side of this equation is the concatenation of the 2q+1 sub-paths.

(ii) l_0 is a $*1$ tangent from S to polygon B_{ij} if this tangent exists in this world.

(iii) for each $j(1 \leq j \leq q-1)$, l_j is $(*j,*j+1)$ common tangent from polygon B_{ij} to polygon $B_{i,j+1}$ if this tangent exists,

(iv) l_q is a $*q$ tangent from polygon B_q to G if this tangent exists, and

(v) for each $j(1 \leq j \leq q)$, k_j is the minimum portion (possibly empty) of the $*j$-directed boundary of polygon B_j between the two osculating points of the previous and next tangents if both tangents exist.

(vi) If any of these sub-paths do not exist in the world, the value $\pi(\sigma)$ is undefined.

Now let us consider a tangent sequence in which the same tangent mode occurs more than once in it.

(III) If a tangent sequence σ' is obtained from $\sigma = B^{*1}_{i1},\ldots,B^{*q}_{iq}$, where $q \geq 1$, $\ldots,B_{iq}$ are elements of W, and $*1,\ldots,*q$ are elements of $\{+,-,++,--\}$, by duplicating one of the tangent modes B^{*r}_{ir}, then the interpreted path $\pi(\sigma')$ is the path $\pi(\sigma)$ added by another complete turn of the *r-directed boundary of polygon B_{ir}. An example of this rule is given in Figure 47b.

D. SEARCHING FOR PATHS USING TANGENT SEQUENCES

A path π is called canonical if there exists a tangent sequence σ such that $\pi=\pi(\sigma)$. The following propositions are essential in the shortest path planning problem:

(I) For any path π, there is a canonical path π_0 which is equivalent to π.
(II) A canonical path is the shortest one of paths in an equivalence relation.

Since a canonical path is the locally shortest path in an equivalence relation class, searching for the shortest path from S to G is equal to searching for the shortest one among canonical paths. The variety of paths comes from the variety of tangent modes to polygons. Since the number of osculating points in a world is finite, this problem is a kind of graph search if we preprocess the world for tangents to obtain all of the visible tangents with osculating point information. A variation of Dijkstra's algorithm and the A* search algorithm is appropriate for solving this geometric problem. The following propositions are essential in the shortest path planning and searching process:

(I) If π is the shortest path joining two points in a world consisting of only convex polygons, the path π is loop-free.

(II) Let π be the shortest path joining two points in a world consisting of only convex polygons, and θ_1 and θ_2 the orientations of two consecutive tangents in π sharing a common osculating vertex with a tangent mode of *, * element {+, -, ++, --} (Figures 48 and 49). Then

$$\theta_1 < \theta_2 \text{ if } * = + \quad \text{and} \quad \theta_2 < \theta_1 \text{ if } * = -. \tag{69}$$

Therefore, when extending a partial path π it is necessary to consider all possible tangent modes from the last polygon, but not from the last osculating vertex of π.

E. USING PATH AREAS TO RESOLVE MULTIPLE LANDINGS

The search for an optimally shortest path through a polygonal world along tangent lines resembles a search through a directed graph for a shortest path from one node to another. The primary difference is that there are two ways for a convex polygon or convex subpolygon to become an osculating polygon for a partial path; through either a counter-clockwise or a clockwise landing of π on the polygon. Thus, a single polygon can equate to two nodes of the search graph, a plus and a minus mode path node. Path planning with m convex polygons is similar to conducting a graph search with $2m$ nodes when the graph is complete.

Another property which differs from the graph search analogy is that there are multiple osculating vertices for a given polygon mode, whether plus or minus. A node in a graph has all directed edges emanating from or terminating at a single point in the node. Thus, it is possible to have two separate partial paths, π_1 and π_2, which osculate on a polygon in the same mode but have different osculating vertices (Figure 50 and 51).

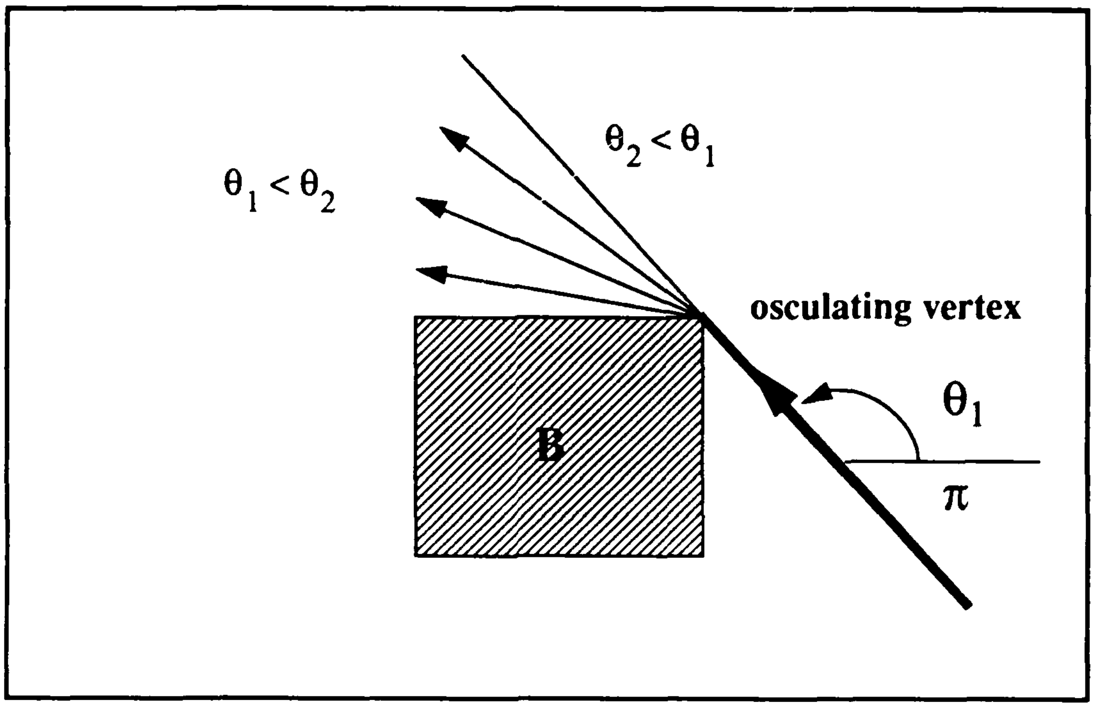


Figure 48 - Example of Allowable Tangent Expansions in Plus Mode

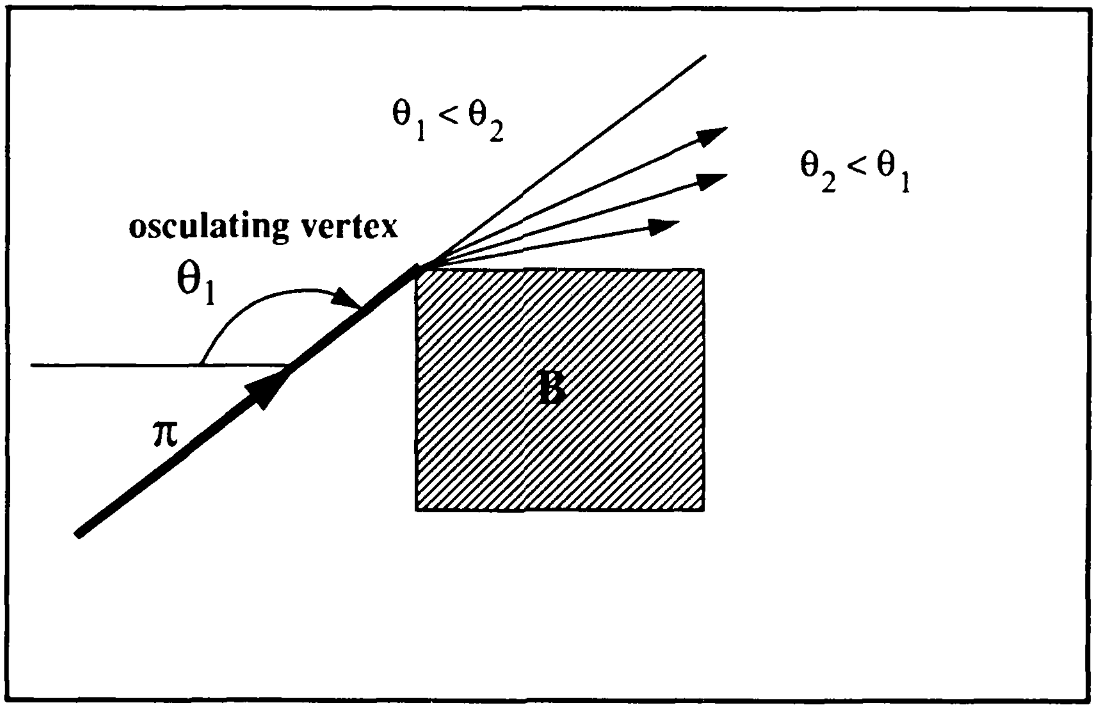


Figure 49 - Example of Allowable Tangent Expansions in Minus Mode

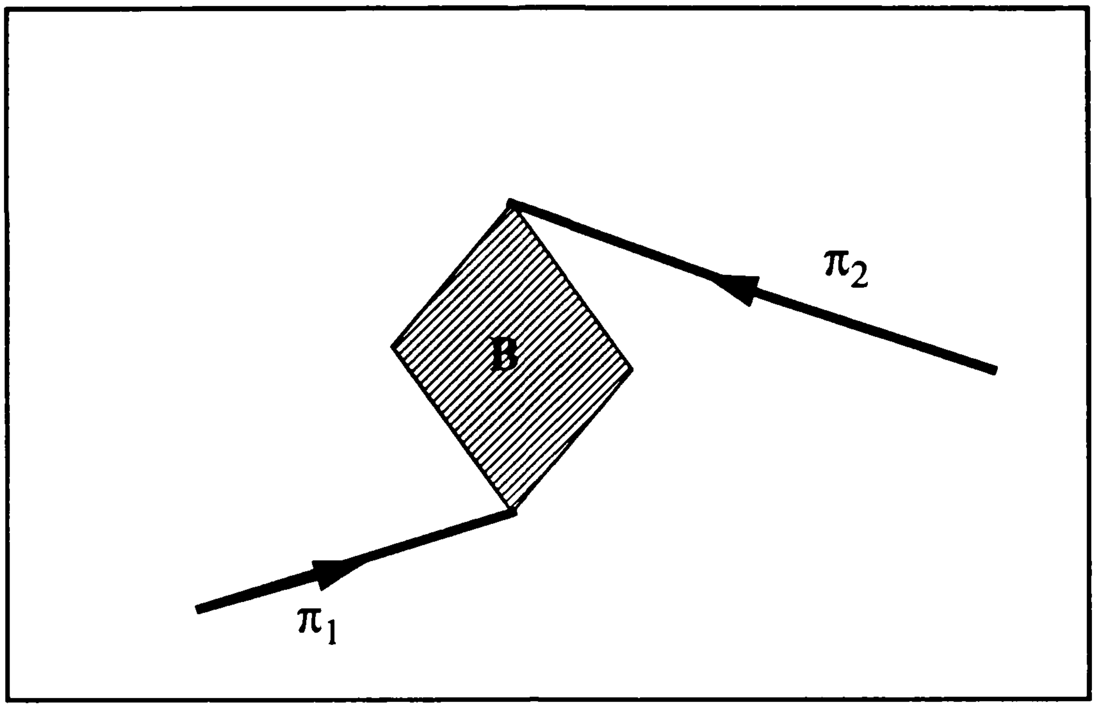


Figure 50 - Two Partial Paths Osculating in Plus Mode on a Single Polygon

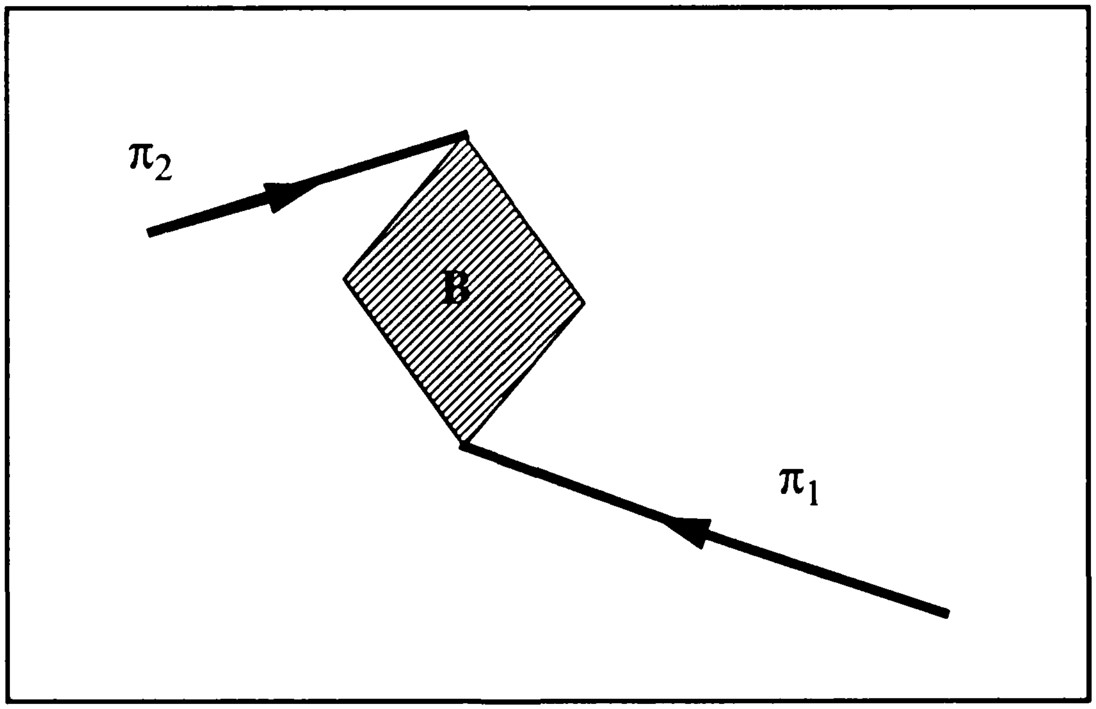


Figure 51 - Two Partial Paths Osculating in Minus Mode on a Single Polygon

When this situation arises, comparison of the respective path lengths does not provide an indication as to which path is more lands on the polygon mode in a more optimal manner.

Consider two partial paths π_1 and π_2 of the same landing mode which osculate on different vertices of the same polygon. It is possible to reach either of the two osculating vertices from the other by traversing along the polygon's edges in the direction corresponding to the path landing mode (Figure 52). Σ_1 is the edge distance which must be traveled by π_1 to reach the osculating vertex of π_2, and Σ_2 is distance π_2 must traverse to reach π_1's osculating vertex. If we reverse the labeling of π_1 and π_2, we simply reverse the relationships of Σ_1 and Σ_2 to the two partial paths (Figure 53). Visually, the problem solution is obvious as we can observe the relative spatial relationships between the two partial paths and make the determination as to which path appears to land first. Computationally, however, the solution is not obvious. The partial paths are specified only in terms of osculating points and tangent orientations, which provide no indication as to the spatial relationships involved. This problem of determining path landing relationships is known as the path *upstream/downstream* landing problem. When two partial paths osculate on different vertices of the same polygon landing mode, the partial path which makes its landing first is said to be the *upstream* path, while the other is said to be the *downstream* path. By computing the closed area under each partial path, we can determine the relative upstream/downstream relationship of two paths.

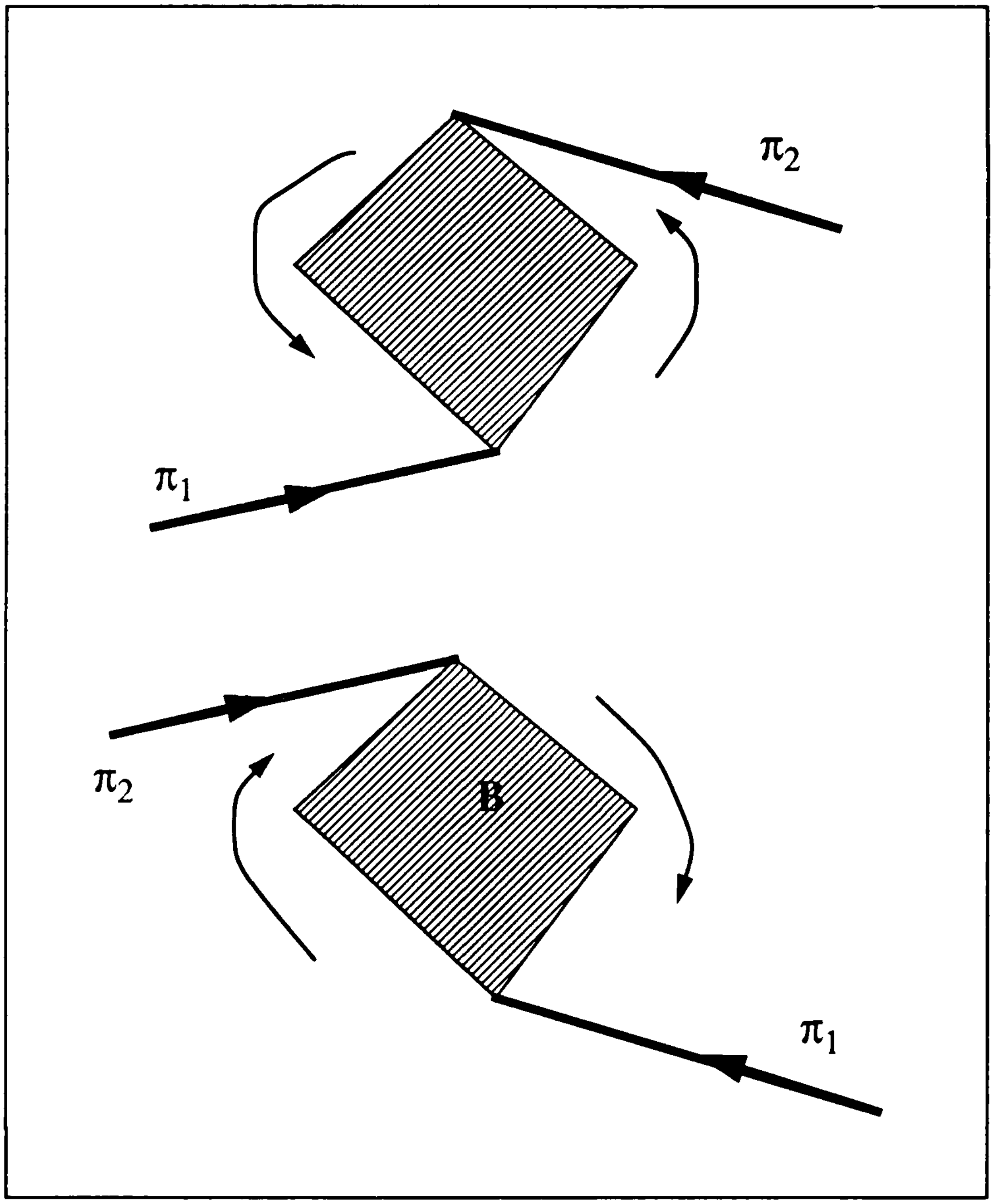


Figure 52 - Traversals to Reach Osculating Vertices

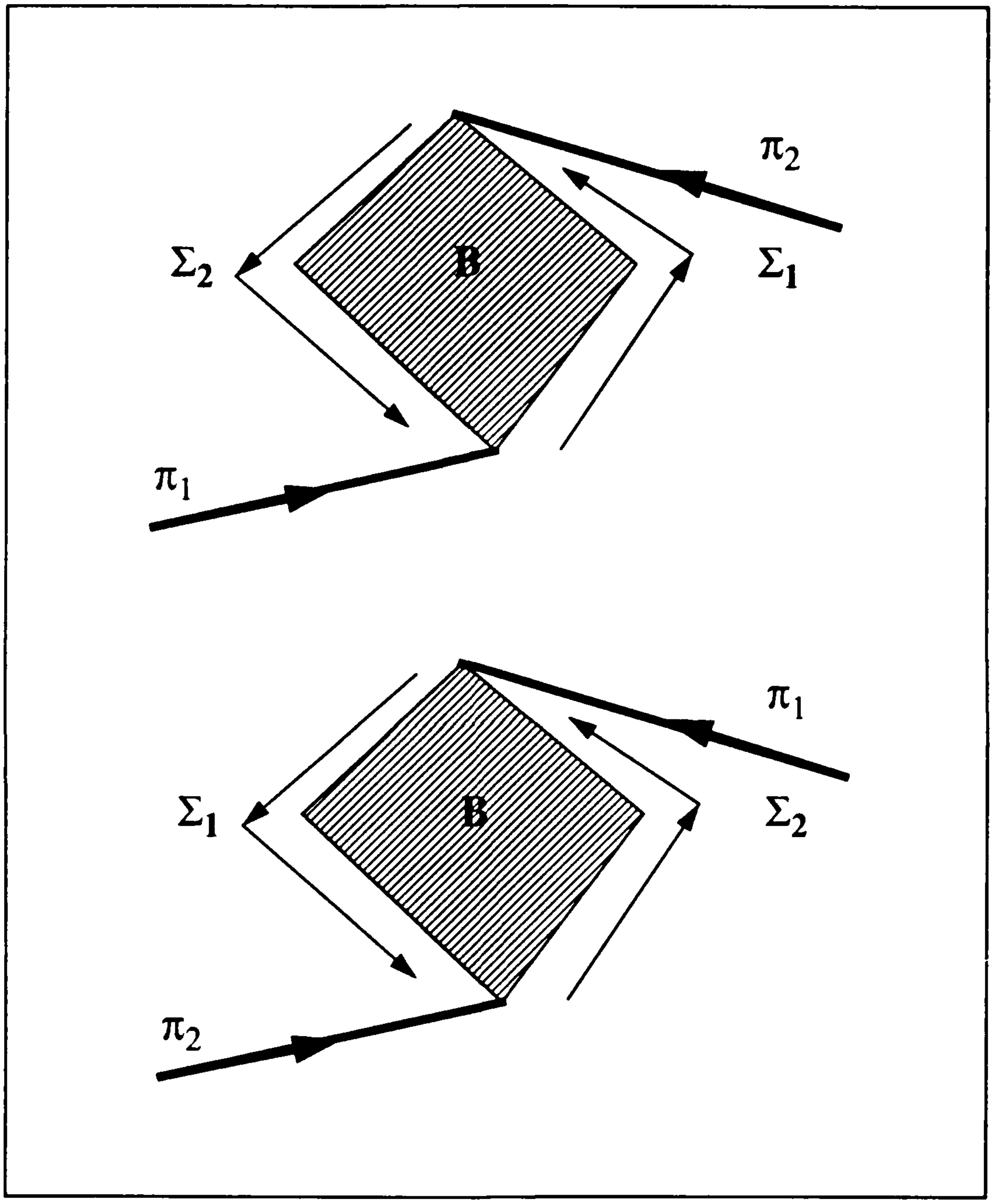


Figure 53 - Impact of Partial Path Label Reversals on Edge Traversals

The formula for determining the spatial relationship of three ordered points was previously introduced as the function $S(p_1, p_2, p_3)$. This same formula will also allow us to compute the area contained within the edges of the triangle formed by the three points. $S(p_1, p_2, p_3)$ will now be referred to as **Area**(p_1, p_2, p_3), where

$$\boldsymbol{Area}(p_1, p_2, p_3) \equiv \begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} \qquad \textbf{(70)}$$

$$= \tfrac{1}{2}\,[x_2 y_3 + x_3 y_1 + x_1 y_1 - x_1 y_3 - x_2 y_1 - x_3 y_2] \qquad \textbf{(71)}$$

$$= \tfrac{1}{2}\,[(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)]. \qquad \textbf{(72)}$$

Recall that the area computed using this formula can be positive or negative, depending on the relative orientation of the three points to each other. Now consider a polygon B with *n* vertices (Figure 54). We can calculate the area of B by fixing a point S in space and summing the area of all of the triangles formed by S and the vertices B when taken in order (Figure 55). Thus,

$$\boldsymbol{Area}(S, B) = \boldsymbol{Area}(S,v_0,v_1) + \boldsymbol{Area}(S,v_1,v_2) + \ldots + \boldsymbol{Area}(S,v_{n-1},v_0), \qquad \textbf{(73)}$$

yields the area of the polygon B.

Calculating the area under a partial path is accomplished in a similar manner, summing the areas of the triangles formed by the partial path osculating points. Given a partial path specified by a series of points, $\pi = (S, p_1, p_2, p_3, p_4)$ (Figure 56), the area covered under π from its starting point S to the current end point p_4 is given by

$$\begin{aligned}\boldsymbol{Area}(S, \pi) &= \boldsymbol{Area}(S,S,p_1) + \boldsymbol{Area}(S,p_1,p_2) \\ &\quad + \boldsymbol{Area}(S,p_2,p_3) + \boldsymbol{Area}(S,p_3,p_4).\end{aligned} \qquad \textbf{(74)}$$

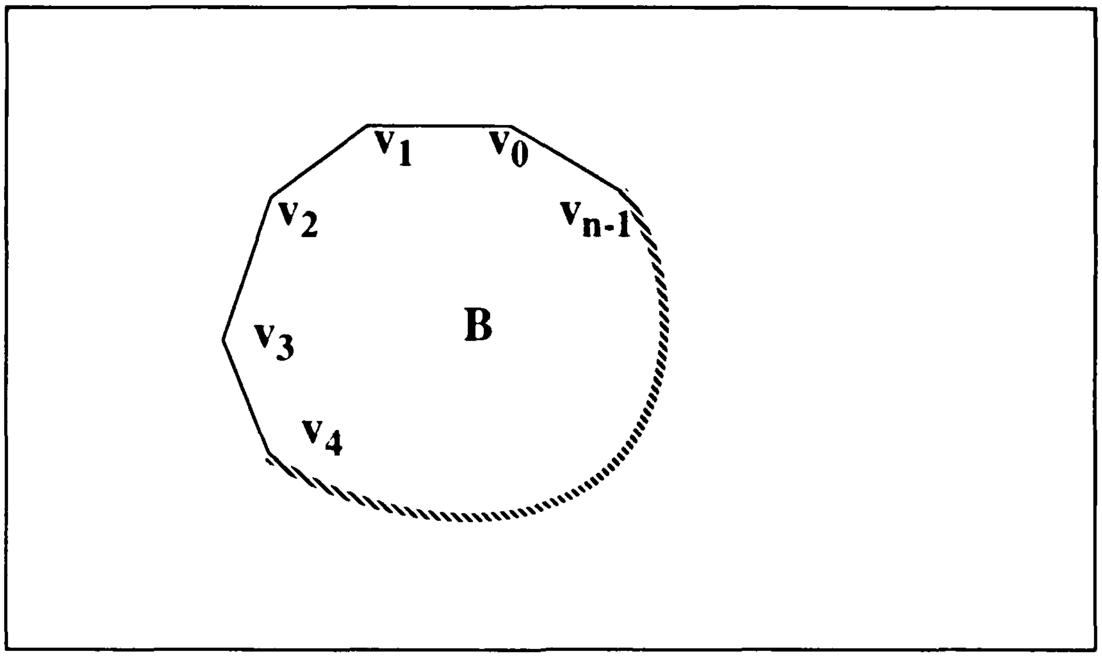


Figure 54 - Polygon with n Vertices

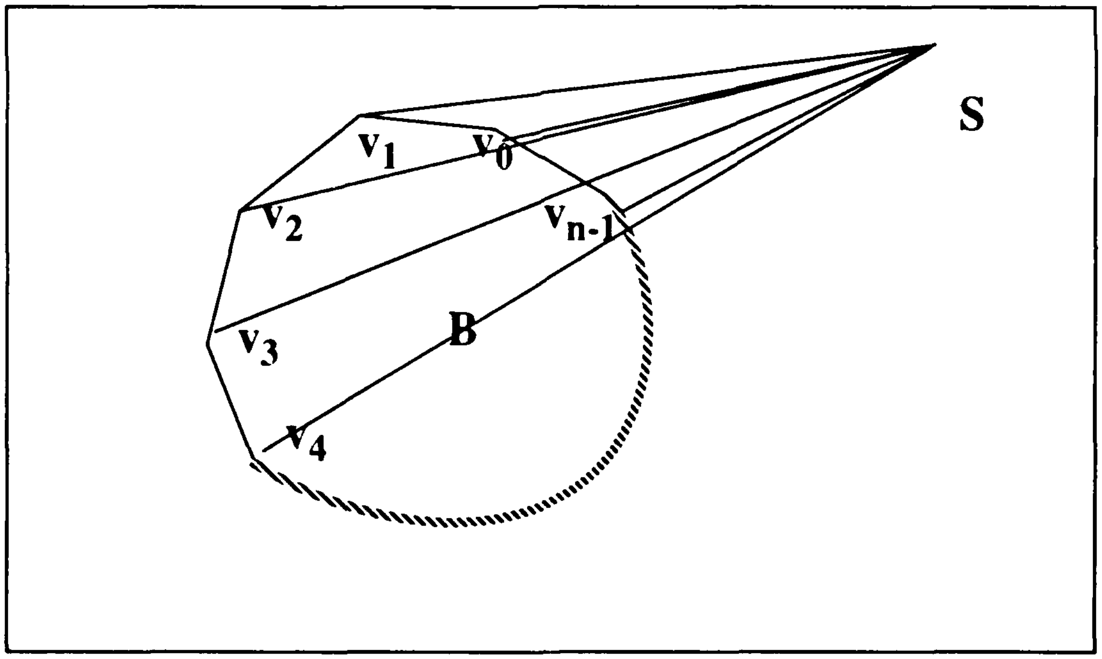


Figure 55 - Computing the Area of B with Triangles

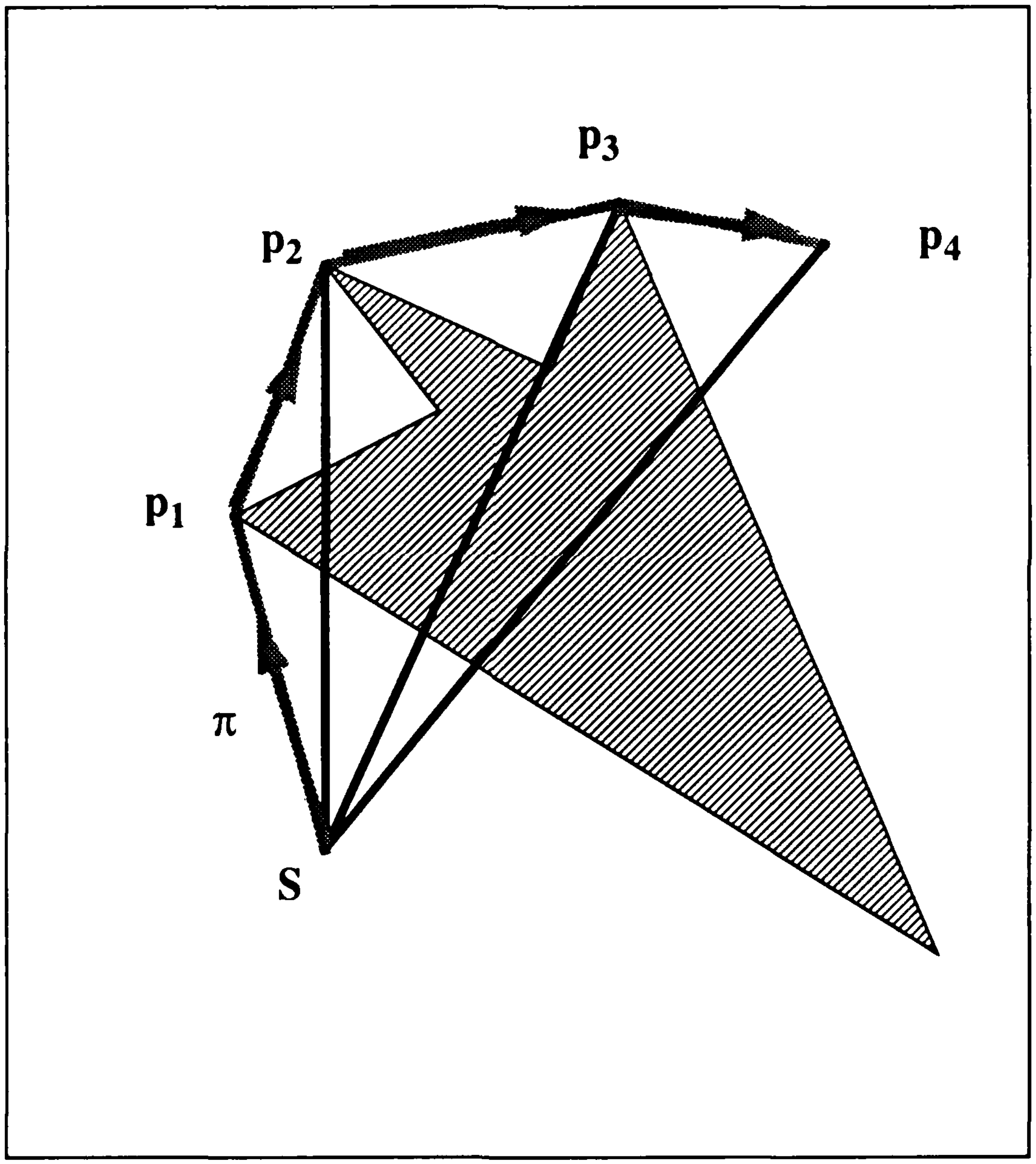


Figure 56 - Computing Partial Path Area from Triangles

We previously indicate that the *Area* function was sensitive to the respective orientations of the three points used in the calculation. This sensitivity to orientation requires that we compare the path areas differently, depending on the landing mode of the partial paths. Consider a world consisting of two convex polygons A and B, and two partial paths π_1 and π_2, which osculate on B in a counter-clockwise mode (Figure 57). The relative partial path lengths are compared using the following criteria:

```
If $Area(S, \pi_2) - Area(S, \pi_1 + \Sigma_1) > 0$ then                                  (75)
    compare the path length of $\pi_2$ to the path length of $(\pi_1 + \Sigma_1)$
else
    compare the path length of $\pi_1$ to the path length of $(\pi_2 + \Sigma_2)$.
```

When the two partial paths π_1 and π_2, which osculate on B in a clockwise mode (Figure 58), the following comparison criteria are used:

```
If $Area(S, \pi_2) - Area(S, \pi_1 + \Sigma_1) < 0$ then                                  (76)
    compare the path length of $\pi_2$ to the path length of $(\pi_1 + \Sigma_1)$
else
    compare the path length of $\pi_1$ to the path length of $(\pi_2 + \Sigma_2)$.
```

IV. CONSTANT CLEARANCE SAFE PATHS

We now have the methods in place for determining shortest paths through a given free space using common tangents and modified graph search techniques. However, the shortest path is the most dangerous one if an error in robot motion control exists even for a point robot. In practical motion tasks in a real world, we need to plan and use safer paths.

The simplest way to do this is to maintain a *constant clearance*, say w_0, at every point on a path and minimize its length under the clearance constraint. When planning paths for a disk robot whose effective radius is w_0, a safe path must maintain a clearance of w_0 from all obstacles to be useful, and these paths consist of straight line segments and circular arcs of radius w_0 (Figure 59). As we reduce the size of w_0, and as w_0 approaches zero, our safe path strip gets closer and closer to becoming the line which osculates on the polygon B, and the disk robot path planning problem resembles that of planning for a point sized robot.

Our goal is to produce safe paths for robot motion, not to reduce a safe path into a point path. The above example, however, illustrates that there exists a degree of similarity between the two extremes. If we can generalize the common tangent path planning concepts for a point, where $w_0 = 0$, and allow w_0 to lie in some arbitrary range of values limited by some k, $0 \leq w_0 \leq k$, then we can formulate a method using *tangent strips* of a fixed width to generate constant clearance paths.

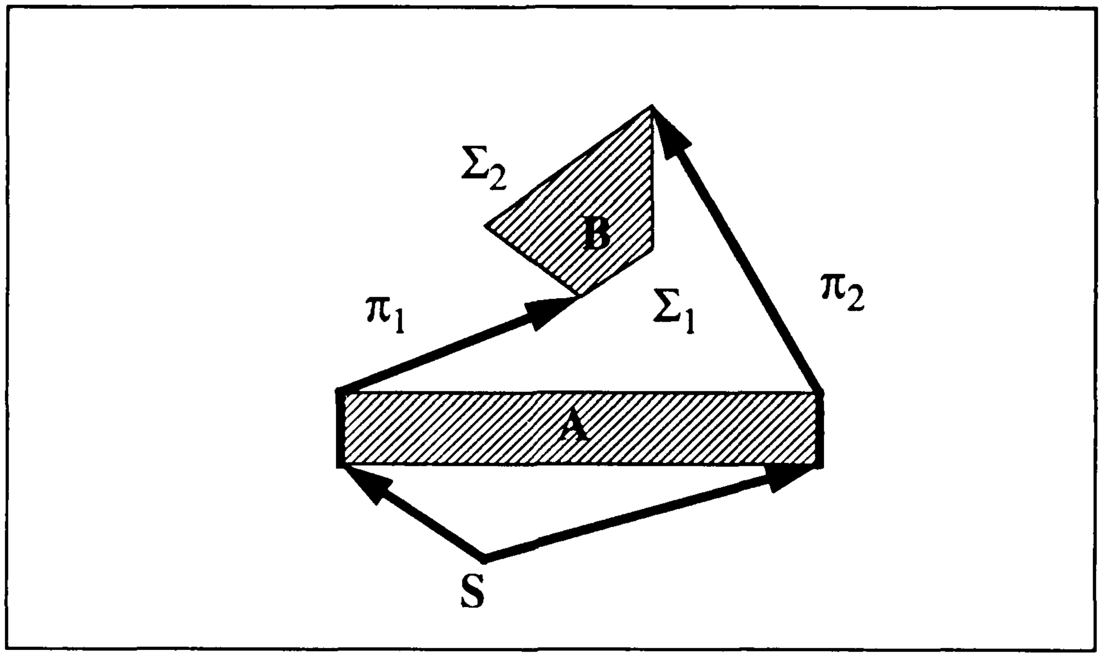


Figure 57 - Two partial Paths Osculating in Plus Mode

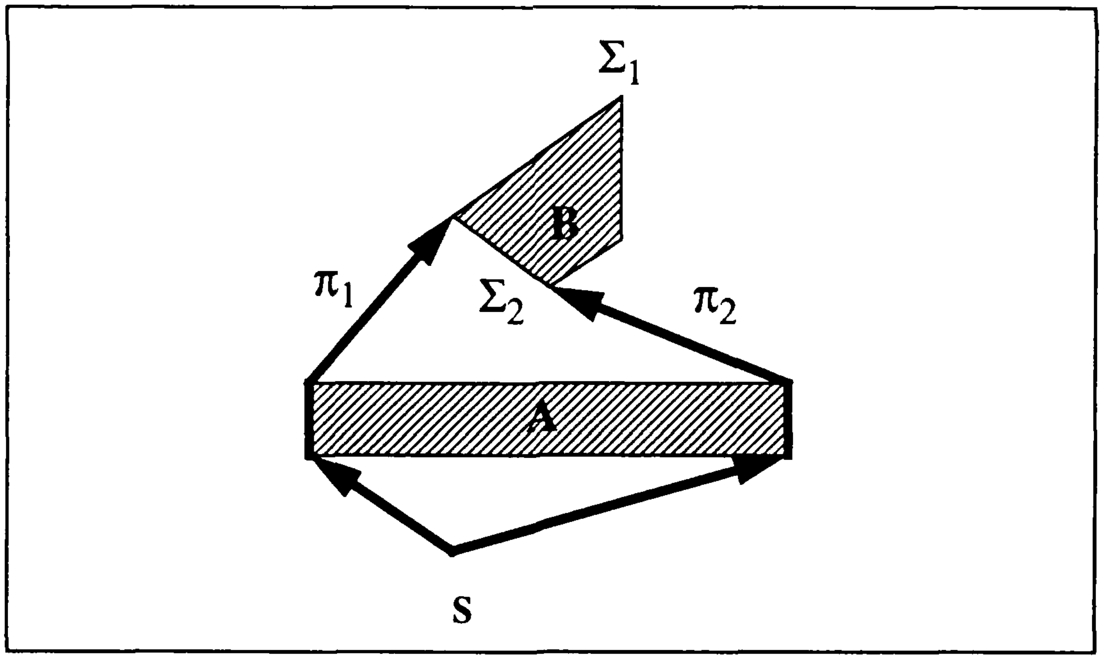


Figure 58 - Two Partial Paths Osculating in Minus Mode

To define a tangent strip of width $2w_0$ from a point p to a polygon B (Figure 60), it is necessary to first construct the tangent line from p to B which osculates on the vertex q. The end points of this tangent are specified by their respective coordinate pairs, $p=(x_p, y_p)$ and $q=(x_q, y_q)$. The direction of the tangent line, β is then used for determining α, the direction of the tangent strip, and where α should be extremum. α is given by

$$\alpha = \beta + \delta, \tag{77}$$

where

$$\beta = \operatorname{atan2}(y_q - y_p, x_q - x_p) \tag{78}$$

and the angle δ between α and β is found using

$$\delta = \sin^{-1}\left(\frac{w_0}{d}\right), \tag{79}$$

where d is the length of the tangent line from p to B. The center point of the end of the tangent strip, r, lies at a distance w_0 from the vertex q, and its coordinates,

$$r = (x_r, y_r) \tag{80}$$

are given by

$$r_x = x_q + w_0 \cos\alpha \tag{81}$$
$$r_y = y_q + w_0 \sin\alpha. \tag{82}$$

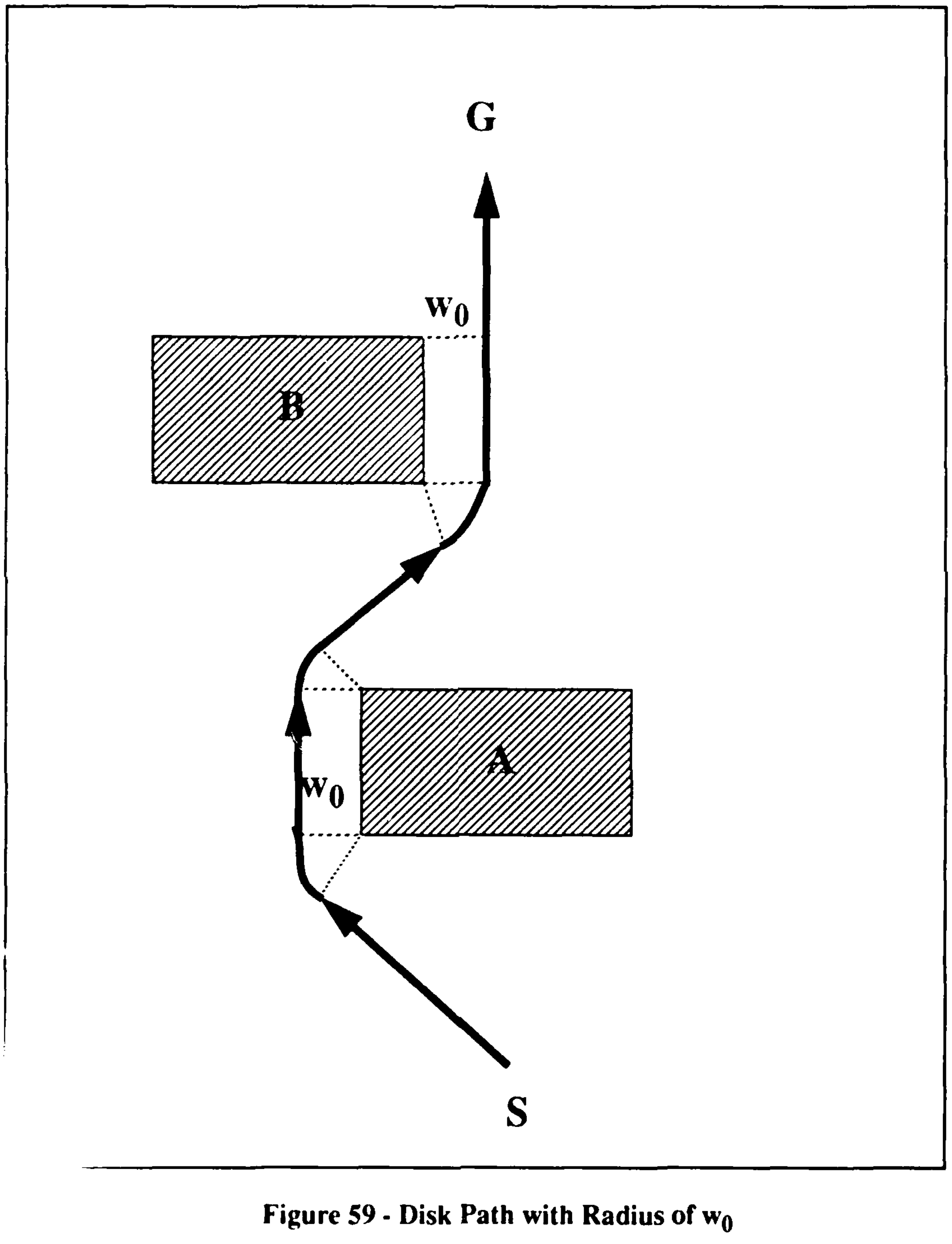


Figure 59 - Disk Path with Radius of w_0

It is also necessary to define a tangent strip between two polygons, A and B, which follows the common tangent osculating on the vertices q_1 and q_2 respectively. Calculating tangent strips when the polygon modes are the same, plus-plus or minus-minus, is straightforward (Figure 61). When the tangent modes are opposite, that is plus-minus or minus-plus, the direction alpha of the strip is calculated through the angle δ by

$$\delta = \sin^{-1}\left(\frac{2w_0}{d}\right), \tag{83}$$

and the distance d is the length of the tangent line from q_1 to q_2. The values of the corners of the tangent strip, r_1 and r_2, are calculated as above, and the center of the tangent strip, c_2, between q_2 and r_2 is given by

$$c_2 = (x_{c2}, y_{c2}), \tag{84}$$

$$q_2 = (x_2, y_2), \tag{85}$$

and

$$x_{c2} = x_2 + w_0 \cos\left(\beta - \frac{\pi}{2}\right) \tag{86}$$

$$y_{c2} = y_2 + w_0 \sin\left(\beta - \frac{\pi}{2}\right). \tag{87}$$

The values for r_1 and c_1 can now be calculated in the same manner.

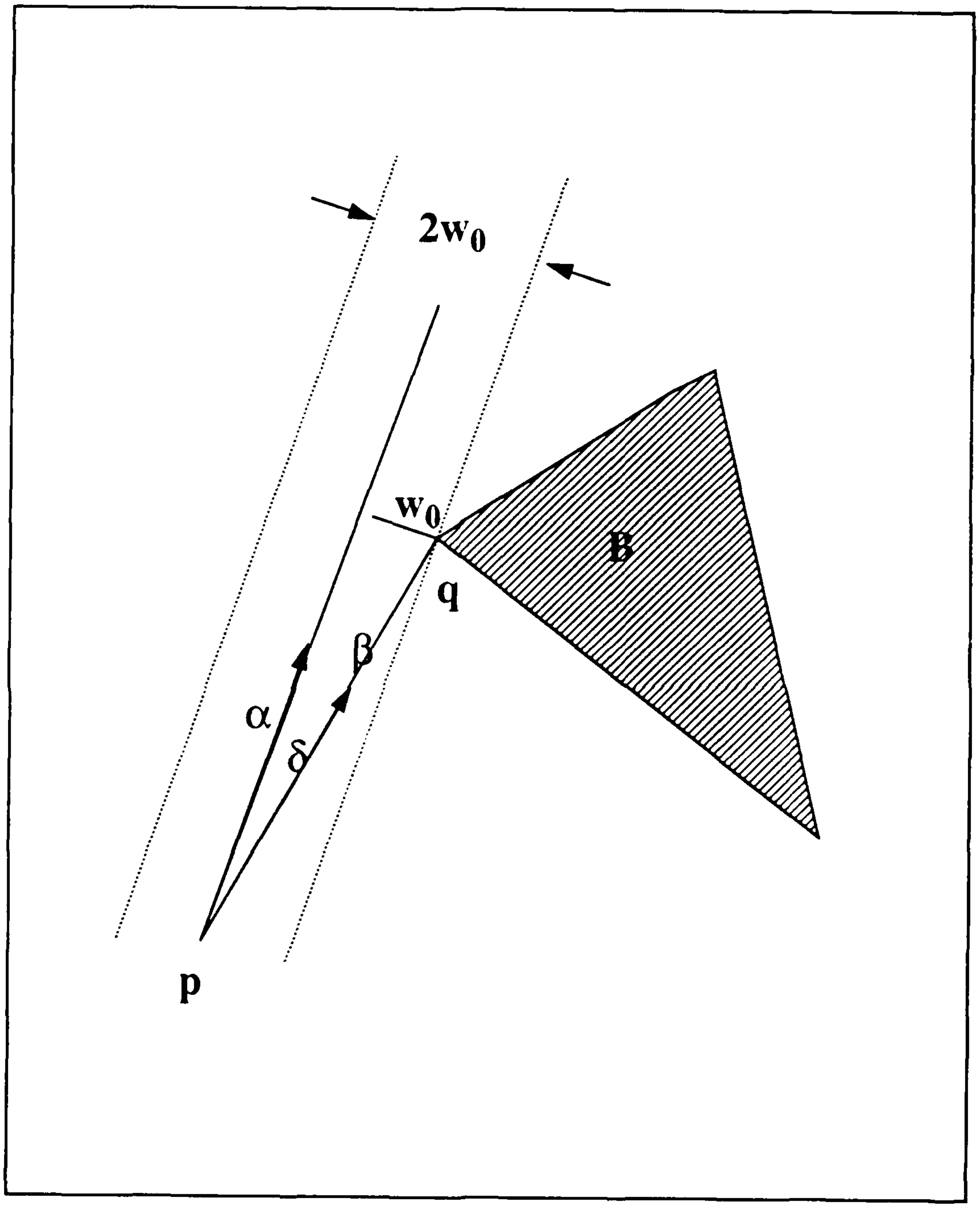


Figure 60 - Tangent Strip From Point to Polygon

The validity of the path formed by the tangent strips is easily confirmed by checking the visibility of the four edges of the strip, and verifying that the circular arcs connecting two consecutive tangent strips is also visible. The interpretation of tangent sequences into paths for a disk robot is similar to that for a point robot. Paths obtained for a disk robot using tangent strips connected by circular wedges possess an advantageous property of tangent direction continuity, which paths for a point robot did not.

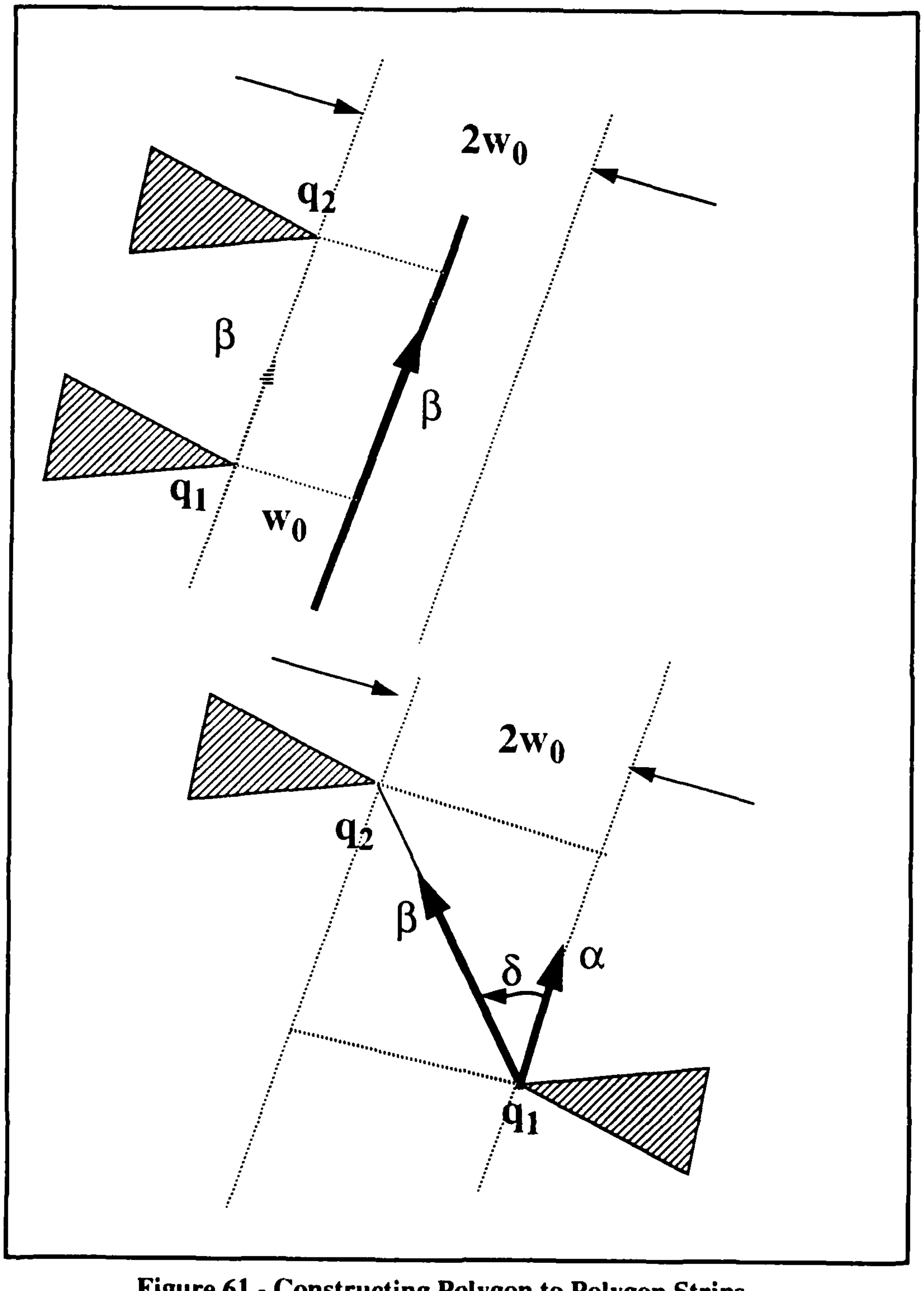


Figure 61 - Constructing Polygon to Polygon Strips

V. IMPLEMENTATION AND RESULTS

A. PROGRAMMING LANGUAGE AND HARDWARE DETAILS

The emphasis for this application was on producing a working prototype with efficiency a second priority. This was the first major coding effort by the author in Lisp, and there was a concurrent goal of learning basic Lisp programming skills while developing a working world model based on the concept of common tangents between obstacles.

Although the programming for this application was done using Allegro Common LISP and the Common Lisp Object System (CLOS), we have not taken an "object-oriented" approach in development of the Lisp code to support this project. We frequently refer to classes and objects in the following discussion, but the claim is not made that this is an object-oriented application. The majority of the functions and methods in the application can clearly be coded more efficiently using another programming language like C, but the capability of Lisp to operate in the interpreted mode made Lisp, in the author's opinion, a better choice for the prototyping process.

The implementation was written entirely in Allegro Common LISP release 4.0 for Sun 4 workstations and SPARCstations running SunOS 4.0 or later. Allegro Common LISP release 4.0 is a significant, yet not complete, move towards Common Lisp as specified by the ANSI X3J13 committee.

A graphical display utility was developed using Allegro Common Windows on X release 2.0. Although not required to run the application, the graphical displays enhance the user's understanding of the path searching process using common tangents. Since the graphical display portion of the application is built around Allegro

Common Windows, it is necessary to have version 11 of the X window system running (preferably release 4.0).

We assume the reader has a basic understanding of the functions and methods peculiar to both Common Lisp and CLOS.

B. BASIC DATA STRUCTURES

The object classes used in this implementation parallel both the theoretical and geometrical aspects of a world consisting only of points, vertices, polygons, and common tangents. The design and implementation was from the bottom up, providing each object class with the greatest possible functionality, and relying as little as possible on information contained at the next higher and lower levels. This results in duplication of data at several levels, however the complexity of the application and the nature of the implementation language made this an advantage during development. Additionally, there were several unresolved aspects concerning references to concave polygons and convex subpolygons at critical development points which made some duplication of data a necessity.

1. Point Class

The *point class* is the most elementary of the object classes used, and is necessary to distinguish between vertices of polygons and simple points within the world. The two most evident such points are the "starting-point" and "ending-point" for the path planning operations. The *point class* slot definition is as follows:

```
(defclass point ()
   ((x-coord :initarg :x-coord       :accessor   x-coord)
    (y-coord :initarg :vertex-type   :accessor   y-coord)
    (name    :initarg :predecessor   :accessor   name)))
```

A brief description of the slots and their purposes are contained in Table 1.

TABLE 1. POINT CLASS DATA STRUCTURE

Component	Description
x-coord	The integer or decimal value of the x component of the Cartesian coordinates of the point in the plane.
y-coord	The integer or decimal value of the y component of the Cartesian coordinates of the point in the plane.
name	Slot reserved for providing a name or designator for a specific xy coordinate pair.

2. Vertex Class

The *vertex class* is the primary structure used in the application for almost all operations. It is from the information stored in the slots of the vertices of the polygons that the type of the polygon is ascertained, tangents are constructed, and where determinations are made on how and what type of partial path expansion is made. The class declaration is shown below:

```
(defclass vertex     ()
          ((coordinates      :initarg   :coordinates      :accessor   coordinates)
           (vertex-type      :initarg   :vertex-type      :accessor   vertex-type)
           (predecessor      :initarg   :predecessor      :accessor   predecessor)
           (successor        :initarg   :successor        :accessor   successor)
           (plus-tangents    :initarg   :plus-tangents    :accessor   plus-tangents)
           (minus-tangents   :initarg   :minus-tangents   :accessor   minus-tangents)
           (parent-name      :initarg   :parent-name      :accessor   parent-name)
           (parent           :initarg   :parent           :accessor   parent)
           (sub-polygon      :initarg   :vertex-type      :accessor   vertex-type)
           (vertex-number    :initarg   :vertex-number    :accessor   vertex-number)))
```

The specific slots and a brief description of their purpose is found in Table 2.

The *predecessor* and *successor* slots contain reference to the neighboring vertices, and are treated as pointers in a doubly linked list, and point to the vertex before and after the current vertex. These pointers allow easy traversal forward and backward, or more appropriately, counterclockwise and clockwise, around the obstacle. The use of pointers, particularly in this LISP based application, permits fast and efficient access to adjacent vertices in either direction without regard to a specific position of this vertex in a list of vertices.

TABLE 2. VERTEX CLASS DATA STRUCTURE

Component	Description
coordinates	Slot contains an instance of the point class, recording the x and y coordinates, as well as a name if assigned.
vertex-type	Slot contains the value, as a string, of either "interior" or "exterior." This corresponds to the two vertex types, concave or convex respectively.
predecessor	Slot containing a reference to the vertex which precedes this one based on the primary rotation direction of counter-clockwise about a filled polygon, and clockwise about a hollow polygon.
successor	Slot containing a reference to the successor vertex, again based on a counter-clockwise or clockwise primary rotation about the polygon, depending on polygon type.
plus-tangents	An unordered list of instantiations of the tangent-line class for all plus tangents osculating on this vertex.
minus-tangents	An unordered list of instantiations of the tangent-line class for all minus tangents osculating on this vertex.
parent-name	String value consisting of the name of the polygon or convex subpolygon to which this vertex belongs.
parent	Reference to the parent polygon for this vertex.
sub-polygon	A "nil" value indicates a convex parent, otherwise slot references the parent convex subpolygon.
vertex-number	Slot contains an integer ≥ 0 indicating this vertex's relative position in the counter-clockwise ordering of vertices of either convex polygon or convex subpolygon, whichever applies.

The *plus* and *minus-tangents* slots each contain an unordered list of the respective tangent types found which osculate on this vertex. Self-tangents, if the polygon is a concave polygon, are stored in these slots and are not differentiated from common tangents to any other polygon. This lack of distinction allows the procedures used in shortest path planning algorithms to treat all tangents the same, not requiring special methods or functions for processing.

The last four slots in Table 2 evolved as the result of a need to distinguish between vertices belonging to a simple convex polygon, and those which were a part of a convex subpolygon. The implementation is not eloquent, but serves an important purpose. When a vertex belonging to a convex subpolygon is examined, it is possible to identify both the parent concave polygon and the parent convex subpolygon by selectively accessing these slots. The *parent* slot always references the original polygon, regardless of whether the parent is convex or concave, and is used in the same manner as a pointer to the parent polygon. The *parent-name* slot has two possible string values. First, it can contain the name of the convex polygon which contains this vertex, or second, it can contain the name of the convex subpolygon which contains this vertex when the parent slot references a concave polygon. The *sub-polygon* slot is similar to the parent slot, except that it is used only when this vertex belongs to a particular convex subpolygon, and contains the traditional LISP value of "nil" when the parent is a convex polygon.

3. Polygon Class

The *polygon class*, briefly described in Table 3, is designed to represent both convex and concave polygons as the same type of object. This makes it possible to develop methods to process all polygons in a similar manner. The class declaration is shown below for reference.

```
(defclass polygon  ()
     ((name                  :initarg  :name                :accessor  name)
      (type-of-polygon       :initarg  :type                :accessor  type)
      (vertice-list          :initarg  :vertice-list        :accessor  vertice-list)
      (exterior-vertice-list :initform ()                   :accessor  exterior-vertice-list)
      (number-of-vertices    :initarg  :number-of-vertices  :accessor  number-of-vertices)
      (xy-coordinates        :initarg  :xy-coordinates      :accessor  xy-coordinates)
      (plus-tangents         :initarg  :plus-tangents       :accessor  plus-tangents)
      (minus-tangents        :initarg  :minus-tangents      :accessor  minus-tangents)
      (convex-sub-polygons   :initarg  :sub-polygons        :accessor  sub-polygons)
      (plus-mode-path        :initarg  :plus-mode           :accessor  plus-mode)
      (minus-mode-path       :initarg  :minus-mode          :accessor  minus-mode)))
```

The slot used to differentiate whether a polygon is convex or concave is the *type-of-polygon* slot. When the polygon is convex, the *type-of-polygon* slot contains the string value "convex" and the *convex-sub-polygons* slot value is unassigned or "nil." If the polygon is concave, *type-of-polygon* slot is assigned the string value "concave" and the *convex-sub-polygons* slot will contain an ordered list of the convex subpolygons into which the concave polygon has been partitioned. Each convex subpolygon consists of an instantiation of the class *convex-sub-polygons*, discussed in the subsequent section.

TABLE 3. POLYGON CLASS DATA STRUCTURE

Component	Description
name	Slot reserved for providing a name or designator for individual polygons.
type-of-polygon	A string value equal to either "convex" or "concave" which corresponds to the polygon's type.
vertice-list	An ordered list of all of the polygon's vertices. Each element of the list is an instantiation of the *vertex* class.
exterior-vertice-list	Also an ordered list of the polygon's vertices, however each vertex in this list is a convex vertex, labeled as an "exterior" vertex in this application.
number-of-vertices	An integer value corresponding to the total number of vertices belonging to the polygon.
xy-coordinates	Ordered list of the xy coordinates of all vertices. Used exclusively for drawing screen displays of the polygon using Allegro Common Windows draw-polygon method.
plus-tangents	A composite list of instantiations of the *tangent-line* class for all plus tangents osculating on this polygon.
minus-tangents	A composite list of instantiations of the *tangent-line* class for all minus tangents osculating on this polygon.
convex-sub-polygons	This slot is used only by concave polygons. Consists of an ordered list of the convex subpolygons making up the original concave polygon. Each element of the list is an instantiation of the *convex-sub-polygon* class.
plus-mode-path	This slot is used only during the path search and expansion phase of the application. The slot is either "nil" or references an instantiation of the path-node class, representing a partial path osculating on this polygon in a plus mode..
minus-mode-path	This slot is used only during the path search and expansion phase of the application. The slot is either "nil" or references an instantiation of the path-node class, representing a partial path osculating on this polygon in a minus mode..

Each polygon has a *name* slot to contain an assigned string value to symbolically represent that particular polygon. The value assigned to this slot can any string value desired. How names are assigned to specific polygons will be addressed later during discussion of data files and program setup.

There are two slots to contain the polygon's vertices; a *vertice-list* slot and an *exterior-vertice-list* slot. Although the nature and purpose of the two slots are different, the individual elements of each list are each instantiations of the *vertex* class.

The *vertice-list* slot is an ordered list of the vertices in a counter-clockwise rotation about the polygon if the polygon is filled, or a clockwise rotation if the polygon is hollow. The first vertex, or v_0, can be any vertex selected at random belonging to the polygon as long as the rotation criteria are followed. Recall that individual vertices have a *predecessor* and *successor* slot, referencing the neighboring vertex on either side. The as a simple list of vertices, the *vertice-list* slot offers two options for processing; each individual vertex can be processed sequentially by moving down the list, or we can access the first vertex on this list, v_0, and use the *predecessor* and *successor* slots to process the vertices in a doubly linked list fashion.

The *exterior-vertice-list* slot is a special purpose slot used only during the tangent construction phase of the application. The list of vertices in this slot are only those vertices which are convex; labeled with the string value "exterior" in this application. The *exterior-vertice-list* slot allows the tangent construction phase of the application to quickly consider only those vertices which are convex. Thus, access to the *vertex-type* slot of each vertex to determine if the vertex is convex or concave and whether tangents can exist is not required. In a world with a large number of complex concave polygons, a modest gain in program efficiency is possible.

The *xy-coordinates* slot is a single purpose, display related slot. Allegro Common Windows is used to provide graphical world display and user interface functionality. The *draw-polygon* method is an Allegro Common Windows built-in function which facilitates fast drawing of polygons, and requires a parameter which is a simple list of x and y coordinates for all vertices in the form $(x_1\ y_1\ x_2\ y_2 \ldots x_n\ y_n)$.

The *plus-tangents* and *minus-tangents* slots are lists containing references to the tangents of the type corresponding to the slot name. The elements of these two lists are simply a collection of the *plus-tangents* and *minus-tangents* slots of all of the vertices making up the polygon. This collection is without regard to whether the polygon is convex or concave. Thus, all of the tangents for a given polygon can be accessed collectively through these two slots, or those of a particular vertex can accessed by locating the vertex and accessing the *plus-tangents* and *minus-tangents* slots of that vertex.

When the polygon is determined to be concave, the application partitions the polygon into convex subpolygons. The convex subpolygons are placed into an ordered list in the convex-sub-polygons slot as an instantiation of the convex-sub-polygon class addressed in the following section. Convex polygons do not use this slot and the value of this slot is the default for the implementation.

The last two slots in the polygon class, *plus-path-mode* and *minus-path-mode*, are used only doing the shortest path search portion of the algorithm, and then only if the polygon is convex. The initial value assigned to these slots is "nil." The slot value remains so until the polygon becomes an osculating polygon in the partial path expansion process. When this occurs, the *plus* or *minus-path-mode* slot is set to reference an instance of the *path-node class*, which contains all of the pertinent information on the current shortest path which osculates on this polygon.

4. Convex-sub-polygon Class

This class is the result of the need to partition concave polygons into convex subpolygons. Comparing Table 3 with Table 4, it is obvious that the convex-sub-polygon class is simply a subset of the polygon class. There is no need to have an *exterior-vertice-list* slot as all vertices in a convex subpolygon are convex. Drawing functions are handled by the parent polygon's *xy-coordinate-list* slot, and since this is the convex subpolygon level, the *convex-sub-polygons* slot is omitted. Otherwise, the slots are named the same and fulfill the same purpose as those in the polygon class. The class declaration is shown below.

```
(defclass convex-sub-polygon   ()
    ((name              :initarg  :name              :accessor  name)
     (vertice-list      :initarg  :vertice-list      :accessor  vertice-list)
     (plus-tangents     :initarg  :plus-tangents     :accessor  plus-tangents)
     (minus-tangents    :initarg  :minus-tangents    :accessor  minus-tangents)
     (plus-mode-path    :initarg  :plus-mode         :accessor  plus-mode)
     (minus-mode-path   :initarg  :minus-mode        :accessor  minus-mode)))
```

TABLE 4. CONVEX-SUB-POLYGON CLASS DATA STRUCTURE

Component	Description
name	Slot reserved for providing a name or designator for individual convex subpolygons.
vertice-list	An ordered list of the convex subpolygon's vertices. Each vertex in this list is a convex vertex, labeled as an "exterior" vertex in this application.
plus-tangents	A composite list of instantiations of the *tangent-line* class for all plus tangents osculating on this convex-subpolygon.
minus-tangents	A composite list of instantiations of the *tangent-line* class for all minus tangents osculating on this convex subpolygon.
plus-mode-path	This slot is used only during the path search and expansion phase of the application. The slot is either "nil" or references an instantiation of the path-node class, representing a partial path osculating on this convex subpolygon in a plus mode.
minus-mode-path	This slot is used only during the path search and expansion phase of the application. The slot is either "nil" or references an instantiation of the path-node class, representing a partial path osculating on this convex subpolygon in a minus mode.

The *plus-tangents* and *minus-tangents* slots are a collection of the tangents osculating only on this convex subpolygon, and the *vertice-list* slot contains only a list of references to the convex vertices making up this convex polygon. The *predecessor* and *successor* slots of the first and last vertex in the *vertice-list* reference vertices which are not in this convex subpolygon. This is an important point to note since, as will be discussed in later sections, the only way to determine when a vertex belongs to this convex subpolygon, another convex subpolygon, or are concave vertices, is to check the individual vertices for parent and type information.

5. Tangent-line Class

The tangent-line class consists of five slots, outlined briefly in Table 5. The first two of which, *end-point-1* and *end-point-2*, are references to the vertices of the polygons or convex subpolygons on which the tangent osculates. The *tangent-type* slot is labeled with a string value, "++", "+-", "-+" or "--", representing the tangent type. The last two slots, *distance* and *angle*, contain decimal values representing the straight line distance between the two ending vertices and the normalized orientation of the tangent line in degrees. The inclusion of the length and orientation of the tangent line is based on the desire to pre-process the world for tangents and retain as much information as possible to expedite the path searching process. When the path searching process is to be repeated on an infrequent basis, these slots may be omitted and calculated only as needed. The class declaration is shown below for reference.

```
(defclass tangent-line  ()
  ((end-point-1     :accessor   end-point-1     :initarg   :end-point-1)
   (end-point-2     :accessor   end-point-2     :initarg   :end-point-2)
   (tangent-type    :accessor   tangent-type    :initarg   :type)
   (distance        :accessor   distance        :initarg   :distance)
   (angle           :accessor   angle           :initarg   :angle)))
```

TABLE 5. TANGENT-LINE CLASS DATA STRUCTURE

Component	Description
end-point-1	Contains a reference to the vertex on which the tangent line first osculates. The slot's value is an instantiation of the class *vertex*.
end-point-2	A reference to the second osculating vertex of the tangent-line. The slot's value is an instantiation of the class *vertex*.
tangent-type	A string value representing the mode of the tangent-line. String values are used to represent the possible tangent modes {++, +-, -+, --}.
distance	This slot contains the decimal value of the straight line distance between the two osculating vertices, end-point-1 and end-point-2.
angle	The value of the angle between the positive x-axis and the tangent-line from end-point-1 to end-point-2.

6. Path-node Class

The path node class is a collection of data contained in other classes which is necessary to record path search and expansion information. Actual instantiations of the path-node class are assigned to either the plus-mode-path or minus-mode-path slots of a convex polygon or convex-sub-polygon. The actual class declaration is shown below, and is summarized in Table 6.

```
(defclass path-node ()
    ((path-mode          :initarg   :path-mode         :accessor  path-mode)
     (landing-vertex     :initarg   :landing-vertex    :accessor  landing-vertex)
     (from-vertex        :initarg   :from-vertex       :accessor  from-vertex)
     (from-polygon       :initarg   :from-polygon      :accessor  from-polygon)
     (from-mode          :initarg   :from-mode         :accessor  from-mode)
     (cost               :initarg   :cost              :accessor  cost)
     (symbolic-path      :initform  :nil               :accessor  path)
     (path-area          :initarg   :path-area         :accessor  path-area)
     (distance-to-goal   :initarg   :distance          :accessor  distance)
     (total-path-cost    :initarg   :total-path-cost   :accessor  total-path-cost)))
```

TABLE 6. PATH-NODE CLASS DATA STRUCTURE

Component	Description
path-mode	Slot contains the string descriptor "plus" or "minus," indicating the mode of the path at this polygon.
landing-vertex	A reference to the specific vertex of this polygon or convex-sub-polygon where the path osculates. The slot's value is an instantiation of the class *vertex.*
from-vertex	This slot references the vertex on the polygon or convex-sub-polygon where the current path to this mode osculated before reaching this polygon. The slot's value is an instantiation of the class *vertex*.
from-polygon	Slot contains a reference to the polygon or convex-sub-polygon on which the partial path osculated before reaching this path-node. The slot's value is an instantiation of the class *polygon* or *convex-sub-polygon.*
from-mode	A string value representing the partial path mode at the from-polygon. Value is either "plus" or "minus."
cost	A decimal value representing the actual path length, to include any edge traversals of polygons, to reach the landing-vertex.
symbolic-path	This slot contains a series of strings representing the symbolic partial path. Example: "start" "A" "+" "B" "-"
path-area	A positive or negative decimal number representing the area under the partial path to reach this polygon path node.
distance-to-goal	A positive decimal value for the straight line distance from the *landing-vertex* to the goal point.
total-path-cost	This slot contains a positive decimal number representing the sum of the *cost* and *distance-to-goal* slots, used as the primary heuristic for selecting a partial path for expansion.

The *path-mode* slot contains the current mode of the partial path. The implementation uses a list of references to instantiations of the path-mode class as the search agenda, rather than creating a separate class to act as a header for each partial path. A certain amount of partial path information is therefore stored in each path node which is used only during selection of potentially best paths for expansion. The *landing-vertex* and *from-vertex* slots hold references to the first and second osculating vertices of the tangent line which osculates on the *from-polygon* and the polygon which containing the instantiation of the *path-node*. The *cost, distance-to-goal*, and *total-path-cost* are slots containing distance information about the relative costs of the partial path. The *cost* slot contains the actual length of the path from the starting point to the path's current end point, which is the vertex referenced in the *landing-vertex* slot. The *distance-to-goal* slot contains the straight line distance from the vertex referenced in the *landing-vertex* slot to the goal point, without consideration as to whether the line is visible or not. The *total-path-cost* slot is the sum of the *cost* and *distance-to-goal* slots, providing a heuristic for the selection and expansion of candidate partial paths during the search process.

The *path-area* slot contains the area under the partial path up to the landing-vertex. As discussed in Chapter III.E, the path area computation is necessary to resolve multiple landings on a polygon or convex subpolygon when this landing occurs at different vertices. Finally, the symbolic-path slot contains a list of string values representing the symbolic progress of the path search and expansion process. It is possible to access this slot value and, following the method defined fc. interpreting tangent sequences, determine the exact route of a partial path without referencing any geometric information regarding the world model.

C. PROGRAM INITIALIZATION

This section defines the format for obstacle data files, explains how to load the program into memory and initialize the application, and explains the initial structuring and processing of the obstacle data. All references here are to uncompiled Lisp files, using "**.lisp**" as a file name extension, which are loaded and executed in the interpreted mode. We confine our explanations to running the application in the interpreted mode, but the program is most efficient when pre-compiled and converted into an executable program.

1. Obstacle Data File Format

Relatively old as far as programming languages go, Lisp remains as one of the better list-processing languages. Rather than developing a complex data input interface using pointers, classes, and structures, the goal was to develop a text based obstacle data file for the application program which could be prepared off-line. We use a simple global variable declaration in the obstacle file consisting of a list of polygons, representing each polygon by a list of vertices. The list of lists format for the obstacle data file provides the potential user(s) with a simple and efficient method for manually entering world definitions. Additionally, this format makes it a simple task to interface this application program with a user designed *world model editor*, permitting the user to interactively create and change world descriptions from a graphics capable terminal. The only requirement from such a world model editor is that its output follow the obstacle data file format detailed in this section.

The world description is prepared by imposing an arbitrary x and y axis on the world, which allows each vertex to be specified as a list of its global x and y coordinates. Each polygon is then composed of a list of vertices belonging to that polygon by selecting an arbitrary vertex as the starting vertex, v_0. A counter-clockwise traversal of the edges is made, listing each vertex until all are included in the list. If the polygon is hollow, possibly for a boundary polygon, the edge traversal is in a clockwise direction around the polygon. Once all polygons have been processed in this manner, the world list consisting of all polygons is composed. This list is entered into the obstacle data file as the global variable ***polygon-list*** using the Lisp declaration *defvar*. An example world declaration is shown below for a world consisting of six polygons:

```
(defvar *polygon-list*   '(((160 140) (340 140) (340 260) (160 260))      <- polygon 1
                          ((400 200) (600 200) (600 400) (500 400)        <- polygon 2
                           (500 300) (400 300))
                          ((160 460) (340 460) (340 540) (160 540))       <- polygon 3
                          ((560 460) (640 460) (640 540) (560 540))       <- polygon 4
                          ((700 400) (800 400) (800 500) (700 500))       <- polygon 5
                          ((100 100) (100 300) (300 300) (300 400)        <- polygon 6
                           (100 400) (100 600) (400 600) (400 500)
                           (500 500) (500 600) (900 600) (900 300)
                           (700 300) (700 200) (900 200) (900 100))))
```

The assignment of symbolic letters, names, or numbers to reference individual polygons is almost always a requirement. This is particularly helpful when the application is run without the advantage of graphical display. The global variable ***obstacle-names*** is included to support this requirement. The declaration for ***obstacle-names*** is made with a list of strings, where each element of the list represents the desired designator for an individual polygon. The ***obstacle-names*** declaration of the example world model above could be made as follows:

```
(defvar *obstacle-names* (list "A" "B" "C" "D" "E" "F"))
```

2. Loading and Initializing the Application

The application program is initially invoked by loading the file "**setup.lisp**" at the Lisp prompt. A series of files are then auto-loaded, each containing functions and methods which accomplish specific tasks once the obstacle file has been loaded. The first file, "**world-def.lisp**," contains global variable declarations, class definitions, and the functions and methods related to the initial structuring of raw world data. The second, "**tangent-functions.lisp**," contains the functions and methods for tangent processing. Finally, "**concave.lisp**," which contains the functions and methods peculiar to processing concave polygons, is loaded. The program then displays the prompt "ENTER OBSTACLE FILE NAME:" requesting the user to enter the complete name and extension of the file which contains the declarations for the global variables ***polygon-list*** and ***obstacle-names***.

3. Initial Structuring and Processing

Once the obstacle data file name has been entered and the file is loaded, the data is then converted into data structures more representative of the complex geometric relationships involved. The structuring and conversion necessary consists of three phases; first, conversion of xy-coordinate pairs into points and vertices, second, the conversion of lists of vertices into polygons, and lastly, the partitioning of concave polygons into convex subpolygons. In this section, we will only address the first phase which deals with the processing of data immediately following program initialization.

The data structure of the global variable ***polygon-list*** lends itself to a series of calls of the Lisp *dolist* macro for first striping off the polygons one by one, and then the individual xy-coordinate pairs. The current procedure for conversion of the xy-coordinate lists in the obstacle data file is accomplished in two such passes over the global variable ***polygon-list***. During the first pass, each xy-coordinate pair is converted into an instantiation of the *point* class, where the *x-coord* slot gets the value of the first element in this pair and the *y-coord* slot gets the value of the second element. Each individual point object is then loaded into the *coordinates* slot of an instantiation of the *vertex* class, and every xy-coordinate pair of the original global variable ***polygon-list*** is now an instance of the vertex class. Another pass is then made on ***polygon-list*** and the predecessor and successor slots are set to reference the vertices which come before and after to form a doubly linked list which can be traversed in either a clockwise or counterclockwise direction.

The principle function and method calls used to accomplish the data conversion discussed above are shown in Table 7, along with a brief description its purpose.

TABLE 7. DATA CONVERSION FUNCTIONS AND METHODS

First Pass	**Purpose**
convert-polygon-point-list	Iterates down the *polygon-list*, calling *convert-polygon-coordinates-to-vertices* on each element (a list of polygon vertices). When completed, the xy coordinate pairs in the original *polygon-list* are all replaced by instantiations of the *vertex* class.
convert-polygon-coordinates-to-vertices	Recursively converts all of the xy-coordinate pairs of each polygon into instantiations of the *vertex* class with a call to the *make-vertex* method.
Second Pass	**Purpose**
coordinate-conversion	Sequentially processes down the *polygon-list*, calling *link-polygon-vertices* on each list of polygon vertices.
link-polygon-vertices	Sets the slot values for the *predecessor* and *successor* slots by calling *connect-links*. The first and last vertex in the list being processed require special processing, since the *predecessor* of the first element is the last vertex in the list, and the *successor* of the last element is the first element in the list.
connect-links	Employs *setf* to set the slot value for the *predecessor* and *successor* slots to the previous and next vertex.

D. CONVEX AND CONCAVE POLYGONS

1. Processing Polygons - The General Case

The next phase is to convert the lists of vertices representing individual polygons into instances of the polygon class. This conversion takes each list of vertices representing a polygon and creates an instance of the *polygon* class. The list of vertices, each of which represents a single polygon, is placed into the *vertice-list* slot of each polygon instance. The processes treats all polygons equally, with no differentiation yet being made as to whether a polygon is convex or concave, and the *type* slot is set to "convex" for all polygons at this time.

When this process is completed, the global variable ***polygon-list*** consists of a list of composite objects, each an instance of the *polygon* class. We still need to differentiate between convex and concave polygons, and to do this we need to classify the individual vertices of each polygon as to their particular type; either convex or concave.

Classification of the vertices is accomplished using the function *determine-vertex-types* (Table 8). It is necessary to sequentially process down the list of polygons contained in ***polygon-list***, accessing the list of vertices in the vertice-list slot of each polygon. Then, we examine each vertex and test the relative orientation of the three points formed by its predecessor, the vertex itself and its successor using the function *point-position*, which is similar to the *order* function given in Equation 17, Chapter II, Mathematical Basis for Common Tangents. Those vertices which are concave have the string "interior" assigned to their *vertex-type* slots and the polygon is immediately classified as a concave polygon by assigning the string "concave" to the polygon's *type-of-polygon* slot. Convex vertices are added to the polygon's list of convex vertices by adding them to the list in the *exterior-vertice-list* slot.

TABLE 8. INITIAL POLYGON PROCESSING

Function or Method	Purpose
polygon-conversion	Traverses down the lists of vertices in *polygon-list* and creates an instance of the *polygon* class for each sublist in *polygon-list*. Each polygon is instantiated with the *type* slot set to "convex". Also names the individual polygons based on the strings in *obstacle-names*.
determine-vertex-types	After all polygons have been converted into an instance of the *polygon* class, the individual vertices are checked to determine if they are interior or exterior vertices. This is done by a call to *determine-obstacle-vertex-types*. When a vertex is found that is interior (implying a concave polygon) the *type* slot of the polygon is set to "concave". If the vertex is convex, it is added to the list in the *exterior-vertice-list* slot
determine-obstacle-vertex-types	Utilizes a call to the method *point-position* to determine the relative orientation of this vertex and its predecessor and successor vertices. The method *point-position* performs the same function as the *order* function in equation 17, Chapter II Mathematical Basis for Common Tangents.

Once we have completed determining the type of each vertex ("interior" or "exterior") and each polygon ("convex" or "concave"), we can then analyze the concave polygons and generate the required instances of the *convex-sub-polygon* class for tangent construction.

2. Implementation of Convex Subpolygons

Before addressing the partitioning of the concave polygons, we should note that the application program deviates slightly from the theoretical approach in definition of convex subpolygons. During development of this program, we had not completely understood nor satisfactorily solved the complexities surrounding common tangents and the more intricate concave polygon shapes which were encountered. Initially, we attempted to partition or sub-divide concave polygons into convex subpolygons by the set of vertices belonging to the convex hull. This was not successful in the general case, and we continued to search for a simple method which would allow unique identification to all common tangents.

The development of the program was at a critical juncture, and the decision was made to adopt a less than optimal approach towards partitioning concave polygons. We determined that by restricting the size of the consecutive convex vertex sets for any given concave polygon to no more than four vertices each, we could guarantee that every tangent could be symbolically references in a unique way. These restrictions solved the partitioning problem, but required more convex subpolygons to represent a concave polygon than the previously stated theoretical approach. It also resulted a requirement to allow an edge traversal between two convex vertices of the same concave polygon but different convex subpolygons to be classified as a tangent line. This was necessary to insure that the convex portions of the concave polygon could be traversed in a continuous manner.

3. Partitioning of Concave Polygons

The subdivision of concave polygons is reduced to examining the type-of-polygon slot of each element of ***polygon-list*** and processing all polygons having "concave" as a slot value. The method *sub-divide-concave-polygons* accomplishes this partitioning by first calling the *list-adjacent-vertices-together* function. Since we have allowed any vertex to be used as v_0, we do not know the vertex type of the predecessor and successor vertices of v_0. Thus, we need to reorder the list of exterior vertices, keeping all adjacent vertices together before proceeding with the partitioning of convex subpolygons. This is the purpose of calling the *list-adjacent-vertices-together* function. The key methods and functions are briefly describe in Table 9.

TABLE 9. CONCAVE POLYGON PROCESSING

Function or Method	**Purpose**
create-convex-sub-polygons	This function uses the Lisp *dolist* macro to process through the global variable ***polygon-list***, calling the *sub-divide-concave-polygons* method on those which are concave.
sub-divide-concave-polygons	Method first calls *list-adjacent-vertices-together* to reorder the consecutive convex vertices, then partitions into groups of four or less consecutive convex vertices, creating an instance of the *convex-sub-polygon* class for each such grouping.
list-adjacent-vertices-together	Reorders the *exterior-vertice-list* of the polygon , placing all adjacent vertices together.
assign-name	Concatenates an incrementing integer to the name of the parent polygon and assigns this string to the *name* slot of the *convex-sub-polygon*.

The *sub-divide-concave-polygons* method then partitions the vertices into groups of four or less consecutive convex vertices, creates an instance of the *convex-sub-polygon* class, and assigns a name to the instance by concatenating an integer onto the string specified for the name of the parent polygon based on the number of *convex-sub-polygons* instances created. The range of the integers used is *0,1,2,...,n-1*, where *n* is the number convex-sub-polygons required for a complete partitioning of the polygon. Each such instance is then added to the list of convex subpolygons in the *convex-sub-polygons* slot of the parent polygon, and the process is repeated until all concave polygons have been sub-divided.

E. TANGENT PRE-PROCESSING

This application locates all legitimate tangents which can exist before entering into the path searching/finding phase. It is fully possibly to develop the tangent construction process on an as needed basis where tangents are only constructed while the path search and expansion process is on going. We plan on modeling real world environments, however, and intend on reusing the model repeatedly for providing path information to semi-autonomous robot vehicles on a recurring basis. Thus, we made the decision to pre-process the world model and use the Lisp *dumplisp* macro to store the pre-processed world as a type of *tangent visibility graph* or *map*. When a path finding/searching requirement surfaces, the map is loaded in the form of the executable image saved by using calling the dumplisp macro, and the path search algorithm can begin almost immediately.

In developing the application, there are four situations where tangent construction methods and procedures are needed. These are shown in Table 10. We restrict our discussion here to only the first two, self-tangents and common tangents, which are necessary to create the *tangent visibility graph.* We will begin, however, with a discussion covering the implementation of visibility testing, a prerequisite for tangent construction.

TABLE 10. TANGENT CONSTRUCTIONS

#	Type of Construction	Situation Where Required
1.	Self-Tangents	Required for concave polygons only, and consists of tangents between convex vertices of the same polygon.
2.	Common Tangents	Addresses tangents between two different polygons. These can be either simple convex polygons of convex subpolygons.
3.	Point-to-Polygon Tangent	Required from a given *starting point* to convex polygons and convex subpolygons. Necessary to link the starting point into the pre-processed *tangent visibility graph.*
4.	Polygon-to-Point Tangent	This construction is used during the path finding phase to determine if the goal point can be reached from a partial path end point.

1. Visibility Testing

The ability to test the visibility of a line segment is critical to the actual construction of tangents in the application, as well as goal visibility testing. The methods implemented here parallel the discussion of visibility testing addressed in the theoretical background sections of this thesis. A brief review of the functions and methods critical to visibility testing is covered in Table 11. In practice, this is the most used section of code in the application, as each individual tangent line must be validated by visibility testing before we can be assured that the tangent line is legal.

TABLE 11. VISIBILITY TESTING

Function or Method	**Purpose**
check-visibility	Tail recursive method which calls *check-for-polygon-intersection* on the supplied tangent-line, the polygon's *vertice-list* slot, and the obstacle at the head of the supplied list of polygons. If *check-for-polygon-intersection* fails to detect intersection (i.e., returns "intersection"), this method calls itself on the tail of the list. This continues until either an intersection has occurred, or the list of polygons is empty.
check-for-polygon-intersection	Using tail recursion, this function tests the supplied tangent-line for intersection with the head of the supplied vertice-list and its predecessor by calling the *intersection-test* method. Calling stops if "intersection" is returned, otherwise the function calls itself on the tail of the supplied vertice-list.
intersection-test	This method tests for intersection of the supplied tangent-line and the line formed by the two supplied vertices. It relies on four calls to the *point-position* method, computing the relative position of the four sets of three points to each other. Utilizes a series of Lisp *cond* clauses to test for intersection. Returns either "intersection" or *nil*.
point-position	Computes the relative position of three points in the same manner as the *order* function introduced in equation 17, Chapter II Mathematical Basis for Common Tangents. Returns a decimal value which is <0, =0, or >0.

2. Self-Tangents for Concave Polygons

The global variable ***polygon-list***, containing the list of obstacles or polygons, is searched sequentially using the *dolist* macro for those polygons with a *type-of-polygon* slot having "concave" as a value. When such a polygon is found, the *exterior-vertice-list* slot is accessed, and the program begins building tangent lines. Recall that only vertices which are convex can have legal tangent lines, and only convex vertices are in the list contained in the *exterior-vertice-list* slot. This means that we can process all vertices in the *exterior-vertice-list* without checking each vertex type.

Using the first vertex in the *exterior-vertice-list* as a fixed point, the program attempts to construct a tangent to the next vertex in the *exterior-vertice-list*. If the next vertex is adjacent to this fixed vertex, the next vertex in the *exterior-vertice-list* is selected. When a vertex-to-vertex pair is found which is a likely tangent line candidate, a landing and leaving validity test of the segment is performed. This test examines each end of the line segment being considered to insure that the tangent line is valid as far as its relative landing and leaving orientation on the polygon. In short, the test examines the relative position of the predecessor and successor vertices at the landing end of the tangent line segment using the *point-position* function. The direction of the tangent line is then reversed and the same test is performed on the predecessor and successor vertices at the leaving end of the tangent line segment. Based on the results of these two tests, we can verify the validity of this line as a tangent. If the segment qualifies as a legitimate tangent with respect to the landing and leaving tests, the final step to confirm the tangent line segment is to check the visibility of the segment through the world.

Visibility testing is completed as previously indicated. If the segment passes the visibility test, then two instances of the *tangent-line* class is created; one representing the tangent in each direction. These instances are then added to the *plus* or *minus-tangent* slots of the leaving and landing vertices respectively, and the process is repeated until all vertices for this concave polygon have been tested.

The remaining concave polygons are tested in a similar manner until the end of ***polygon-list*** is reached. Table 12 contains a brief description of the principle functions and methods called during this phase. When all concave polygons have been processed, the *dolist* macro is used in a nested form to copy the self-tangents for the concave polygons into the *plus* and *minus-tangents* slots of both the convex subpolygon and the parent polygon.

TABLE 12. SELF-TANGENT CONSTRUCTION FOR CONCAVE POLYGONS

Function or Method	**Purpose**
find-self-tangents	This method sets up the initial call to locate-self-tangents by stripping the first vertex off of the exterior-vertice-list of the polygon.
locate-self-tangents	Attempts to construct a tangent to the first vertex in the exterior-vertice-list. Calls itself recursively on the remaining vertices of the same polygon.
adjacency-test	Simply tests if the fixed-vertex is adjacent to the current-vertex. If they are adjacent, no tangent construction is attempted.
check-line	Simple test of the predecessor and successor vertices at each end of the candidate tangent line for proper tangent positioning.
test-verify-and-attach	Tests the visibility of the candidate tangent line using a call to intersection-test. If the line is valid (no intersections) then the tangent and its reciprocal are added to the plus or minus-tangents list of the fixed and current vertices.

3. Common Tangents Between Polygons

Locating common tangents between polygons is similar in approach to finding the self-tangents of the concave polygons. We must consider possible tangent constructions from one convex vertex to all the other convex vertices of different polygons or convex subpolygons which have not yet been considered. When we find and verify a tangent from one polygon's vertex to the vertex of another polygon (*forward tangent*), we have also verified the that the tangent in the opposite direction (*return tangent*) is valid. This is the *reciprocal relationship* of the *forward* and *returning* tangents discussed in Chapter II.H. and illustrated in Figure 27. Thus, it is not necessary to attempt to verify this return tangent in the opposite direction if we develop our tangent construction process around this *reciprocal relationship* of forward and return tangents.

Construction of common tangents between polygons is implemented as a recursive process in the method *locate-some-tangents*. Each call uses the first polygon in ***polygon-list*** as a fixed reference polygon for the tangent construction process to take advantage of the above relationships, constructing tangents from the convex vertices of the fixed polygon only to the polygons or convex subpolygons remaining in the rest of ***polygon-list***. Construction of the remaining tangents between the polygons in ***polygon-list*** is then accomplished by calling *locate-some-tangents* on the tail of ***polygon-list***. As the recursion progresses deeper, the list of polygons remaining in the rest of ***polygon-list*** becomes fewer and fewer until we reach the point where there are no polygons remaining.

The methods and functions used to perform the tangent construction process are the same or parallel ones developed for constructing self-tangents (Table 13). Tangent candidates are found by selecting the first vertex in the *exterior-vertice-list* slot of the fixed polygon, and attempting to construct a tangent to the vertices in the *exterior-vertice-list* slots of the polygons remaining in ***polygon-list***. Every effort was made to reduce the number of visibility tests that were required for possible tangent lines, since the tangent line requires testing against every polygon edge to confirm visibility. The *check-line* and *point-position* methods are used to analyze the landing and leaving relationships of the ends of the potential tangent line, allowing the determination to be made on whether the line is a viable tangent. Only those tangent lines which meet the tangent landing and leaving criteria in the cond statements of the function locate-all-tangents are tested for visibility. Finally, the valid tangent lines are added to the plus and minus-tangents slots of both the landing and leaving vertices, and the process is repeated for the next vertex in the *exterior-vertice-list.*

TABLE 13. COMMON TANGENT CONSTRUCTION

Function or Method	**Purpose**
locate-some-tangents	Removes the first polygon from ***polygon-list*** and begins the tangent construction process using the vertices of this polygon sequentially as the fixed vertex. Recursively calls itself on the tail of ***polygon-list***.
find-tangents	Using the dolist macro to consider each polygon in the tail of ***polygon-list***, calls *locate-all-tangents* with the fixed vertex and the *exterior-vertice-list* of each convex polygon, or the *vertice-list* of a convex-sub-polygon.
locate-all-tangents	Attempts to construct a tangent from the fixed vertex to each of the vertices in the supplied list of convex vertices. Calls *check-line* to evaluate the landing and leaving relationships, and if valid, calls *test-verify-and-attach* for further processing.
test-verify-and-attach	This method first test the visibility of the potential tangent line. If the line is visible (i.e., the string "valid-tangent-line" is returned from the call to *verify-tangent-line-visibility*) then this tangent is added to the *plus* or *minus-tangents* slot of the leaving vertex, and the reciprocal tangent is added to the *plus* or *minus-tangents* slot of the landing vertex.

F. POINT-SIZED SHORTEST PATHS

Searching for shortest paths proved to be the most complex portion of the application to develop. The functions and methods implemented are both lengthy and complex, and some restructuring and simplification is currently an on-going project. Rather than offer a detailed analysis of the coding involved in the implementation, we will discuss the purpose and functions of the key functions and methods of the implementation, and how the implementation is executed.

The approach for locating the shortest point-sized path was divided the procedure into two initial and four subsequent phases. The two initial phases consist of first, entering the start and goal points and determining if the start and goal are initially visible, and second, if they are not visible, then connecting the start point into the pre-processed *tangent line visibility graph*. Connecting the start point required the construction of the visible tangent lines from the start point to the polygons contained in the global variable ***polygon-list***.

It was necessary to rewrite many of the functions and procedures already developed to handle polygons and convex-sub-polygons, allowing them to now handle a tangent line consisting of a single point and a vertex, versus a tangent consisting of a pair of polygon vertices. The approach is exactly the same as that for constructing common tangents from a single fixed vertex, except that the methods are slightly different. These methods are contained in the file "**tangent-functions.lisp**" and are listed in Table 14 below.

TABLE 14. POINT-TO-VERTEX METHODS

Methods
find-tangents-from-point
construct-tangents-from-point
locate-all-tangents-from-point
check-line-from-point
check-line-from-vertex
intersection-test
attach-tangent

Once a tangent has been found, we want to use this tangent as the basis for a path expansion, rather than simply add it to a plus or minus-tangents slot. The purpose of the path-node class is to fulfill this requirement, allowing the tangents from the start point to be connected to the *tangent line visibility graph* and serve as the first set of nodes for the path expansion process. As a tangent is found, we create an instance of the *path-node* class and make assignment of the slot values of this instance (see Table 6 to review the slot names and class structure). Next we assign the appropriate *plus-path-mode* or *minus-path-mode* slot of the landing polygon or the landing convex-sub-polygon to reference this *path-node* instantiation. Finally, we build a list of references to the *path-node* instantiations in the global variable called ***polygon-mode-list***, which serves as the list of the end points of the active partial paths.

When all tangents from the start point to the surrounding polygons and convex subpolygons is complete, ***polygon-mode-list*** contains a list of the modes of all polygons and convex subpolygons visited from the start point. Note that a polygon or convex subpolygon can be osculated on by these tangents originating from the start point in two possible modes; a plus or counter clockwise mode, and a minus or clockwise mode. Each element of ***polygon-mode-list*** is an instance of the *path-node* class, and the slots of each of these elements contains the information and references necessary to begin the shortest path search.

Searching for the shortest path is now a matter of implementing the concepts introduced in Chapter III, Shortest Paths Using Common Tangents into the framework of Dijkstra's Algorithm. Heuristic control of path end point expansion is integrated into Dijkstra's search, using the evaluation function

$$\mathbf{f(n) = g(n) + h(n),}$$

where *n* is the path-node instance representing a partial path end point, *g(n)* is the actual length of the path from the start point to the partial path end point, and *h(n)* is the straight line distance from the path end point to the goal. Since *h(n)* cannot possibly overestimate the distance to the goal, we can be assured that the search algorithm will find the optimal path to the goal using *f(n)*.

We begin by sorting the *path-node* elements in ***polygon-mode-list*** based on the values in the *total-path-cost* slot of each *path-node*, where the elements with the lower values are placed first. Recall that the value in this slot is the sum of the current path length and the distance from the path end point to the goal point, providing the heuristic control measure for partial path expansion. After sorting, the first element in ***polygon-mode-list*** represents the partial path (an instance of the *path-node* class) with the lowest overall cost. The expansion of this partial path proceeds by first determining whether the last osculating polygon in the path is a convex polygon or a convex subpolygon. This determination is necessary due to the orthogonal nature of the world model.

Partial paths ending with an osculation on a convex polygon require considering only tangents emanating from the osculating vertex and either its successor (if the landing mode is plus or counter-clockwise) or its predecessor (if the landing mode is minus or clockwise). Processing partial paths which osculate on convex subpolygons require the same consideration be given to tangents emanating from the osculating vertex. However, the tangents emanating from all subsequent vertices when traversing the convex subpolygon in the direction consistent with the partial path landing mode must also be considered. Thus, for a convex subpolygon, it is necessary to examine all tangents from counter-clockwise vertices when landing in a plus mode, and all tangents from clockwise vertices when landing in a minus mode. The incorporation of the landing and leaving tangent orientation criteria from Equation 69 restricts the number possibilities for expansion from the osculating vertex, and prevents inclusion of less than optimal local paths.

When the candidate tangent lines are identified to expand the current partial path, there are three possibilities surrounding the new partial path end points generated. First, the polygon or convex subpolygon has not yet been visited. When this situation arises, the particular plus or minus-mode-path slots of the new landing polygon or convex subpolygon have "nil" as a slot value. We need only create a new instance of the path-node class and assign the appropriate plus or minus-mode-path slot to reference this instance.

The second possible finding is that another partial path has already visited this particular polygon or convex subpolygon in the same landing mode and both osculate on the same vertex. Comparing the path lengths of the two partial paths indicates which is the shortest path. The plus or minus-mode-path slot of the polygon or convex subpolygon is set to reference the partial path with the lower path cost, and the other path is discarded.

The last possibility is when the two partial paths osculate on the same polygon or convex subpolygon in the same landing mode but on different landing vertices. This situation requires differentiating between the two partial paths by comparing the relative magnitude of the path areas to reach the same end points. The path areas are computed and compared as discussed in Chapter III.E. Using Path Areas to Resolve Multiple Landings, and the partial path with the greater relative cost is discarded, and the plus or minus-mode-path slot of the polygon or convex subpolygon is set to reference the shorter of the two paths.

With the expansion of each path, the end points of the tangent line which lead to the next polygon or convex subpolygon is tested to determine if the goal is visible or not. When the goal is visible from a tangent end point, the search process is terminated by setting the global variable *goal-found-flag* to "true." This has the effect of immediately stopping all further expansions and terminates path finding process. Optimal path data is then retrieved by examining the instance of the path-node class which led to the discovery of the goal. We can obtain the actual path length and the symbolic path description, which allows the instances of the path-node class to be retrieved from the polygons osculated on during the search process.

H. PATH PLANNER DISPLAY AND OUTPUT

The application developed for this thesis was oriented not towards providing a numerical or computational analysis of another path finding algorithm. The intent was to validate the theoretical work done thus far in applying obstacle common tangents to the optimal path finding task. The output of the implementation is a series of Allegro Common Windows graphical displays which are the result of conducting path finding searches for various start and goal point combinations over two example worlds. Additionally, we can examine the instances of the path-node class contained in the global variable ***polygon-mode-list*** on completion of the search to access path data in greater detail than that provided by the graphical display. The experimental results discussed here refer to the graphical output of the program contained in the appendices.

Appendix A contains the obstacle data file and initial program output for a small world consisting of only a few simple polygons. Appendix B contains similar data for a more interesting and complex world. The world model in Appendix B is indicative of the requirements surrounding path planning in more intricate manmade surroundings.

Within Appendices A and B, the displays are organized to parallel the work performed by the application program. The first figure contains the Lisp code for the example world obstacle data file. The second figure shows the free and filled space of the example world, where black is filled space and white is free space. The third and fourth figures are line drawings of the polygons before and after the partitioning of concave polygons into convex subpolygons. The last two figures in each appendix show self-tangents for concave polygons, and the display of all tangents once the pre-processing is complete.

Appendix C contains figures showing the final output displays for various start and goal point combinations for the two example worlds. These figures show the final expansion phase where the goal point was reached and the search terminated. In addition to showing the shortest path from the start point to the goal point, the display also shows the intermediate partial path expansions to polygons and convex subpolygons examined during the search. These intermediate partial path expansions are the current shortest paths to the plus and minus modes of the respective polygons and convex subpolygons at the time of program termination. They clearly reflect the influence that the heuristic cost function has on the selection of partial paths for expansion.

The results of the application program clearly indicate that optimal path finding using obstacle common tangents is a viable path planning method. The pre-processing of the world model permits the path finding process to become one of conducting a relatively straightforward graph search of the tangent visibility graph. Although we were not speed or efficiency focused in program development, it is interesting to note some informal observations about the actual performance of the application.

Using the two example worlds discussed in Appendices A and B, we were able to obtain final paths almost instantly (between 2 and 9 seconds) without display. Without discounting the rather complex interactions between Allegro Common Windows, the X11 windowing system, and the operating system, we consistently found optimal paths using the Common Windows displays in 20 to 30 seconds, even for the most complex positioning of the start and goal points.

These are informal timing observations rather than accurately measured performance windows, but the observations are worth mention. An efficiently coded application, probably in C, using pre-processed tangent visibility graphs would most likely achieve far superior performance results than this Lisp prototype.

LIST OF REFERENCES

Bochereau, L., Wolfshein, D., and Kanayama, Y., *Simulation of Model-based Path Planning for a Mobile Robot*, unpublished technical report, University of California, Santa Barbara, California, June 1988.

Feinberg, E. B., *Characterizing the Shortest Path of an Object Among Obstacles*, paper contained in Information Processing Letters, vol. 31, no. 5, 12 June 1989.

Hilton, P. *Algebraic Topology*, Courant Institute of Mathematical Sciences at New York University, 1969.

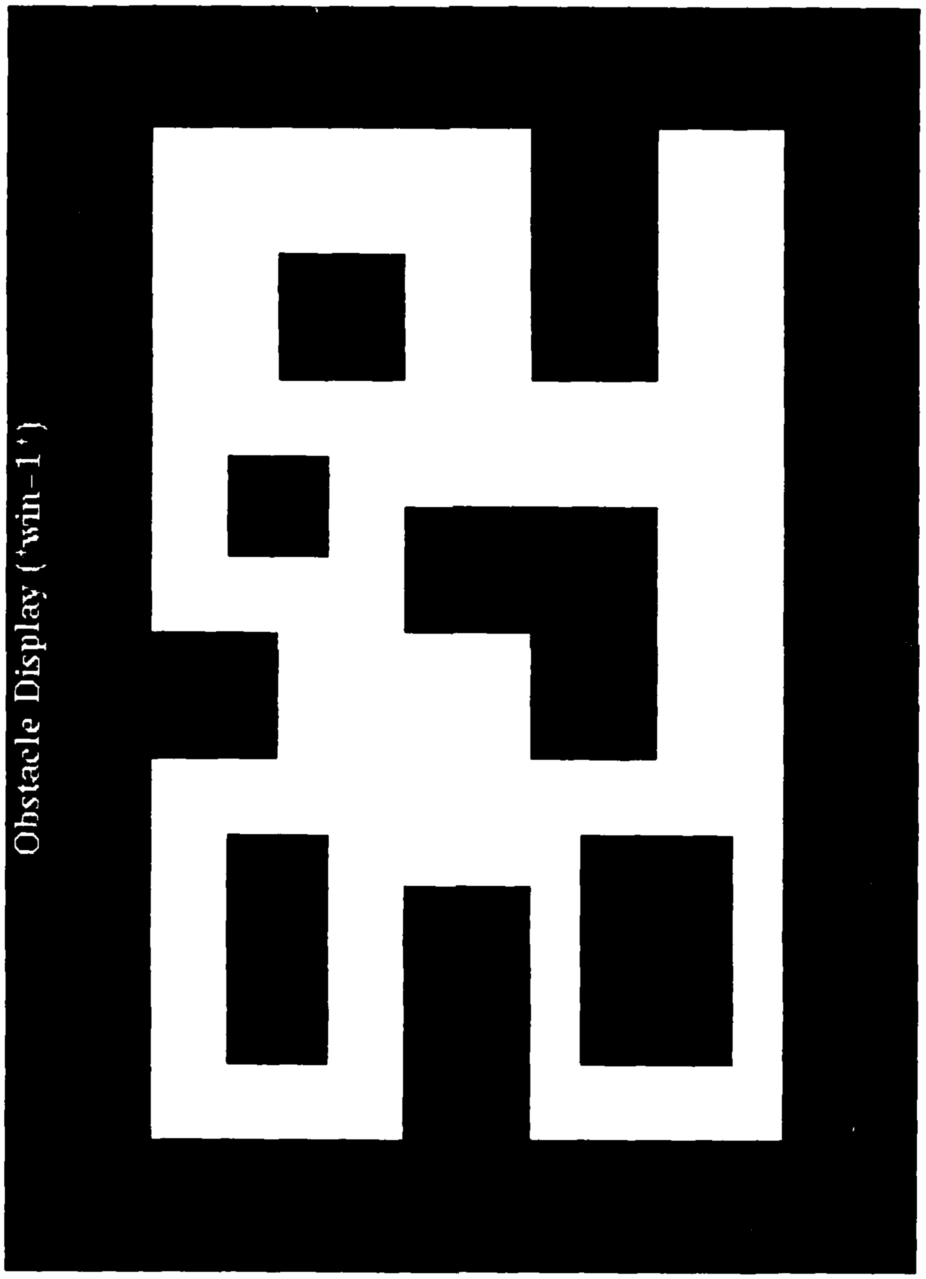
Kanayama, Y., and DeHaan, G. R., *A Mathematical Theory of Safe Path Planning*, unpublished technical report, University of California, Santa Barbara, California, June 1988.

Lozano-Perez, T. and Wesley, M. A., *An Algorithm for Planning Collision Free Paths Among Polyhedral Obstacles*, Communications of the ACM, vol. 22, no. 10, pp. 165-175, 1979.

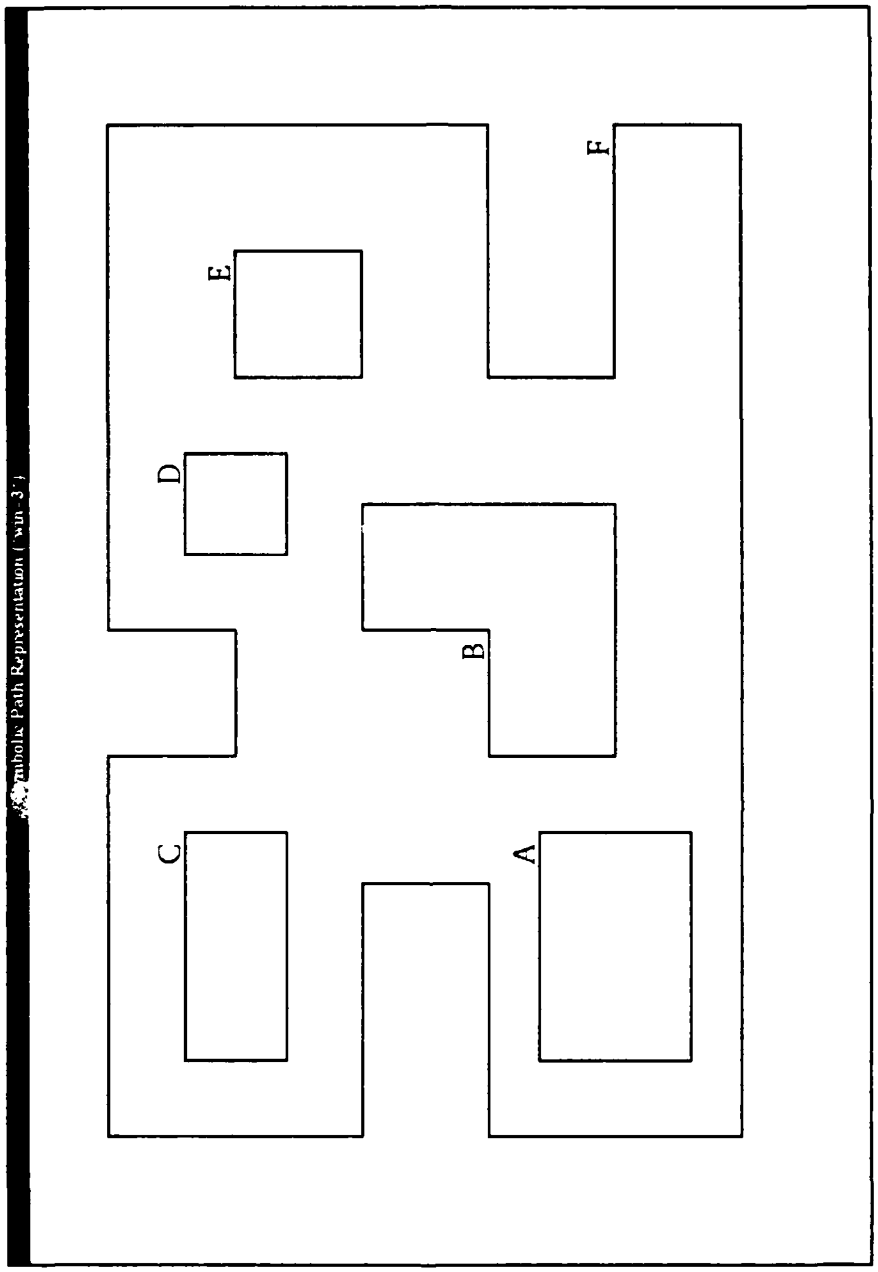
APPENDIX A

EXAMPLE WORLD NUMBER ONE

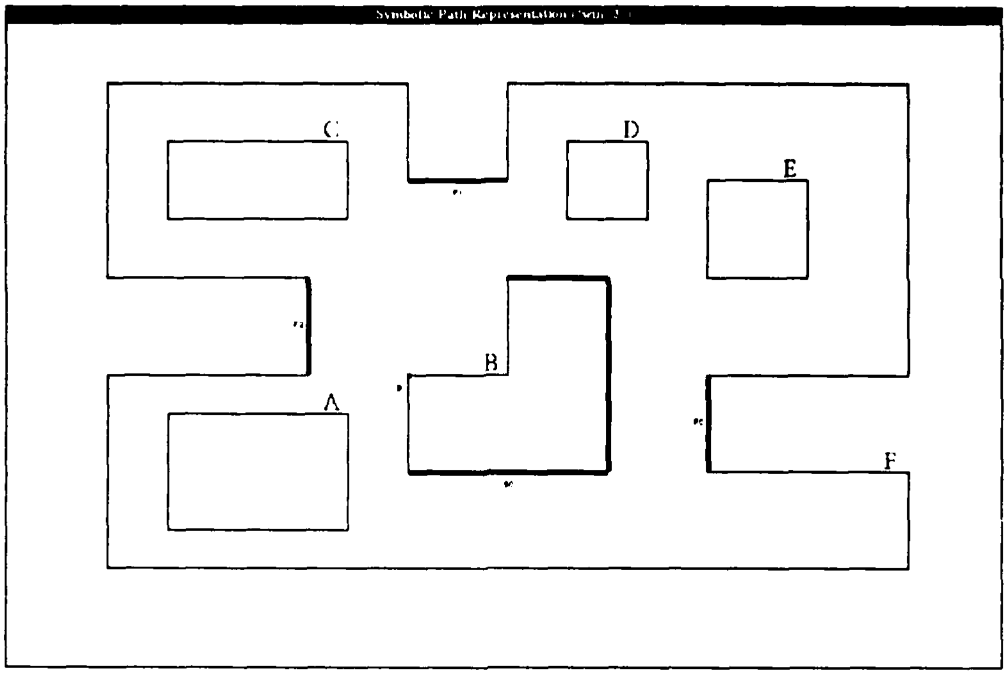
```
(defvar *obstacle-names* (list "A" "B" "C" "D" "E" "F" "G"))

(defvar *polygon-list* '(((160 140) (340 140) (340 260) (160 260))  ; A
                         ((400 200) (600 200) (600 400) (500 400)   ; B
                          (500 300) (400 300))
                         ((160 460) (340 460) (340 540) (160 540))  ; C
                         ((560 460) (640 460) (640 540) (560 540))  ; D
                         ((700 400) (800 400) (800 500) (700 500))  ; E
                         ((100 100) (100 300) (300 300) (300 400)   ; G
                          (100 400) (100 600) (400 600) (400 500)
                          (500 500) (500 600) (900 600) (900 300)
                          (700 300) (700 200) (900 200) (900 100))))
```



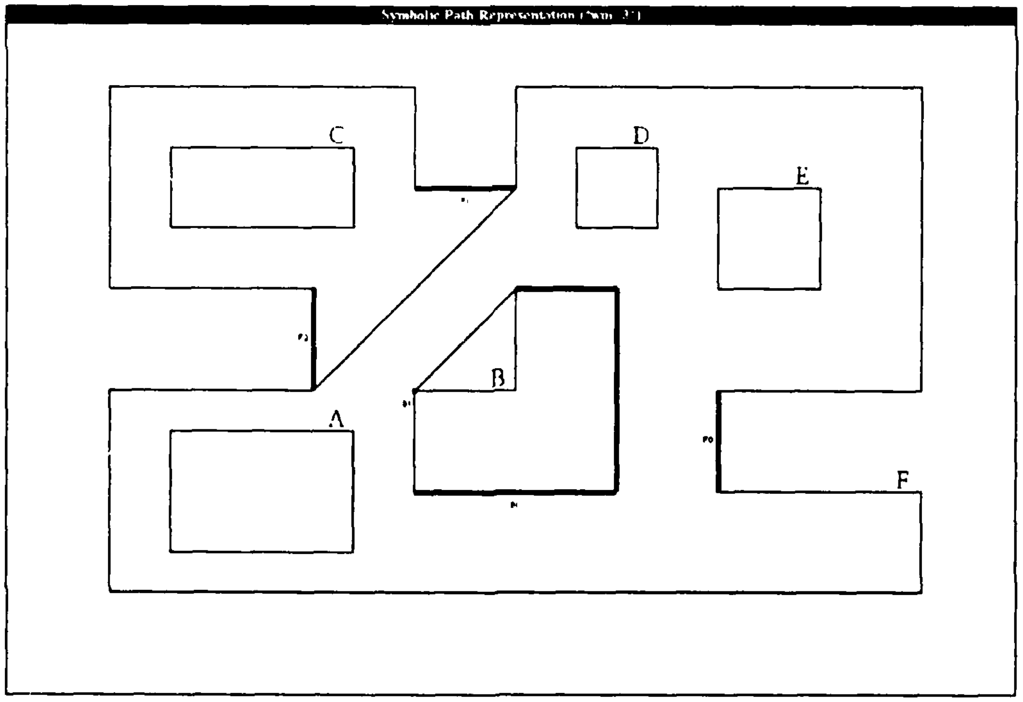
Screen Image of Example World 1 Free and Filled Space



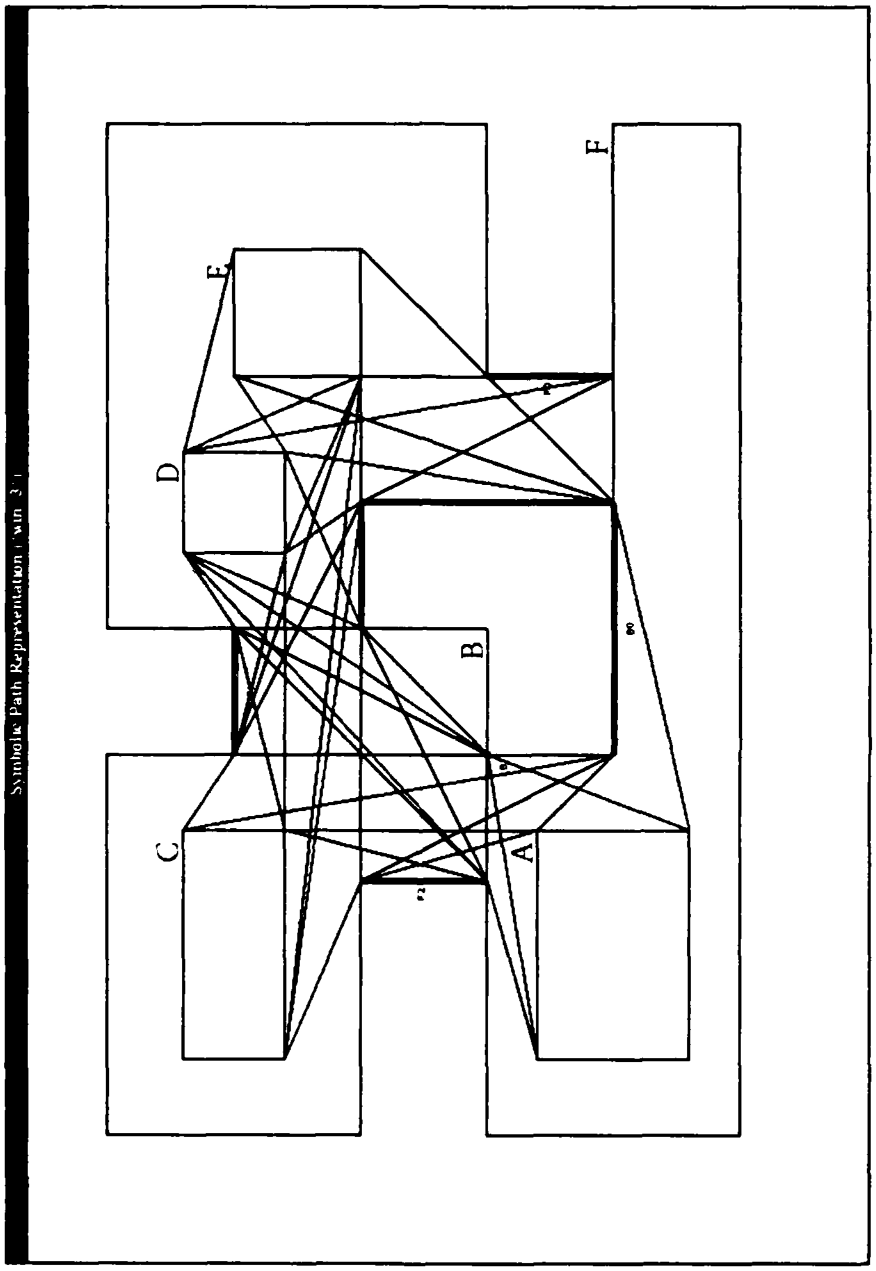
Screen Image of Example World 1 Before Partitioning into Convex Subpolygons



Screen Image of Example World 1 AfterPartioning into Convex Subpolygons



Screen Image of Example World 1 After Self-Tangent Construction

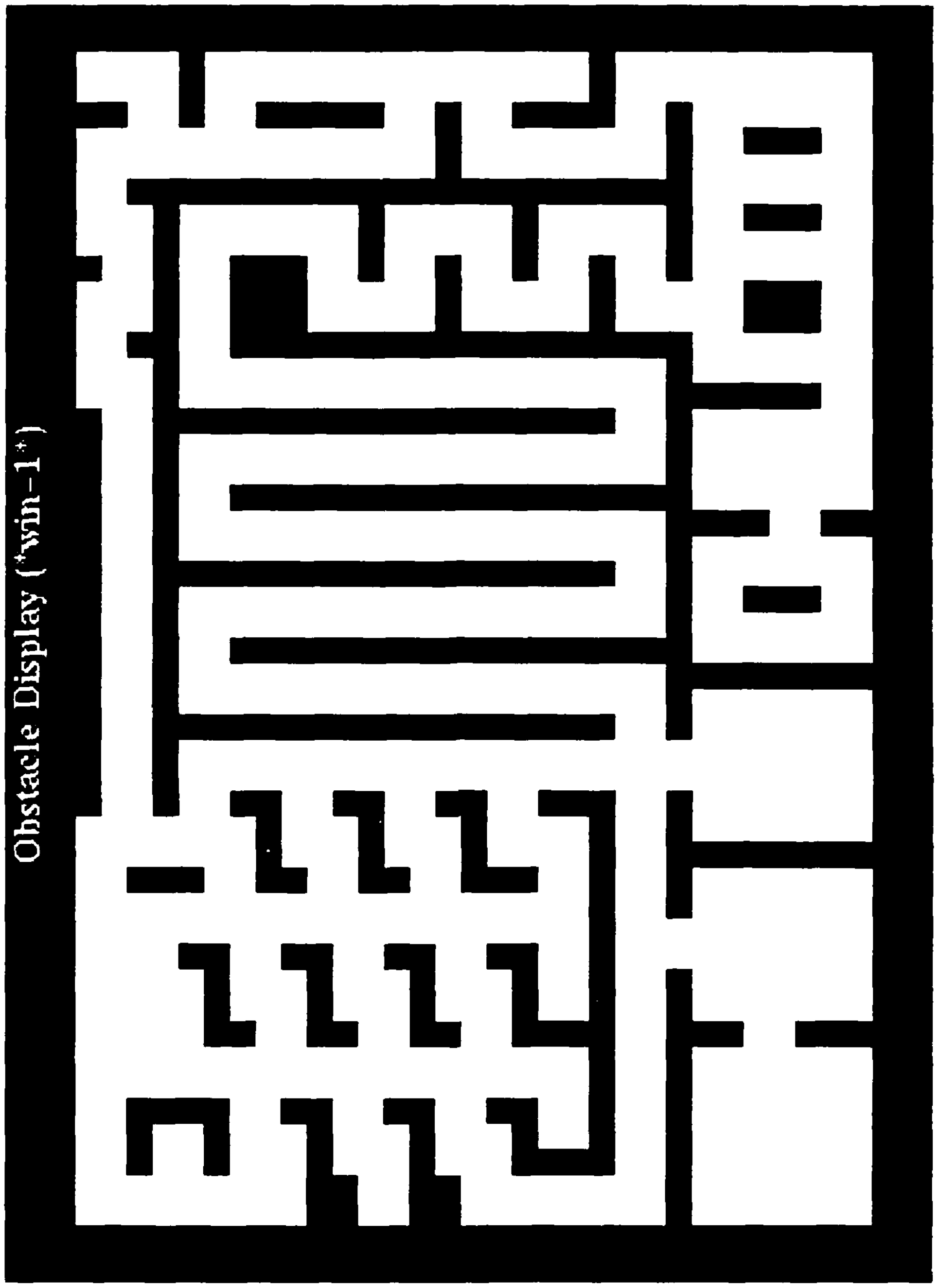


Screen Image of Example World 1 AfterFinal Tangent Construction

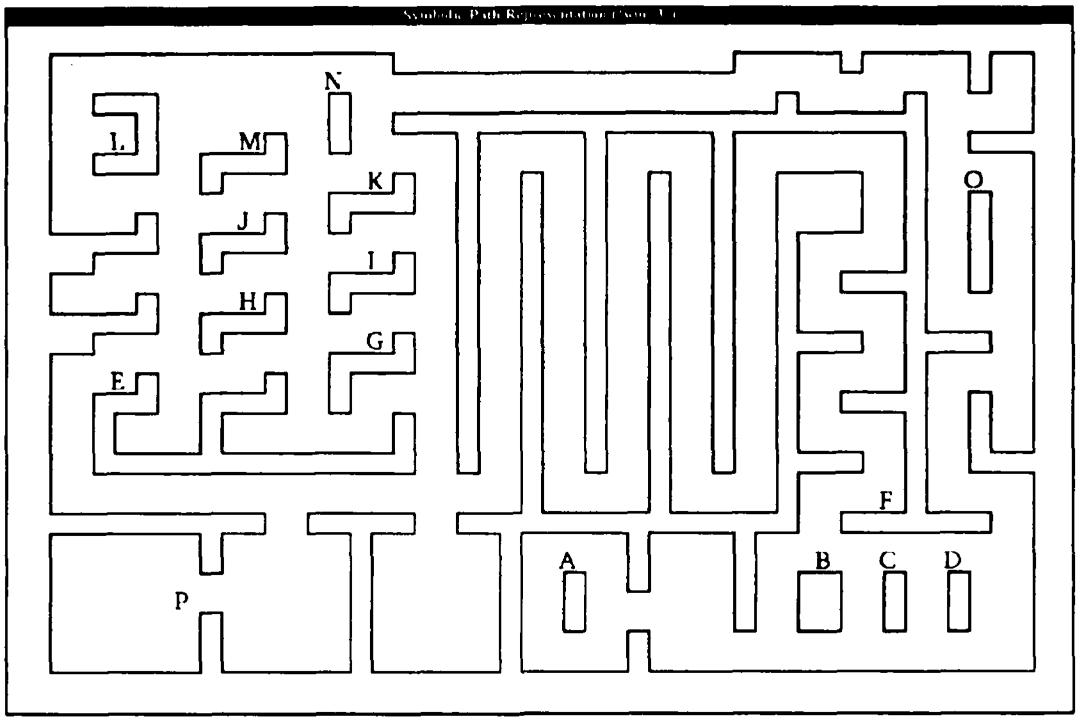
APPENDIX B

EXAMPLE WORLD NUMBER TWO

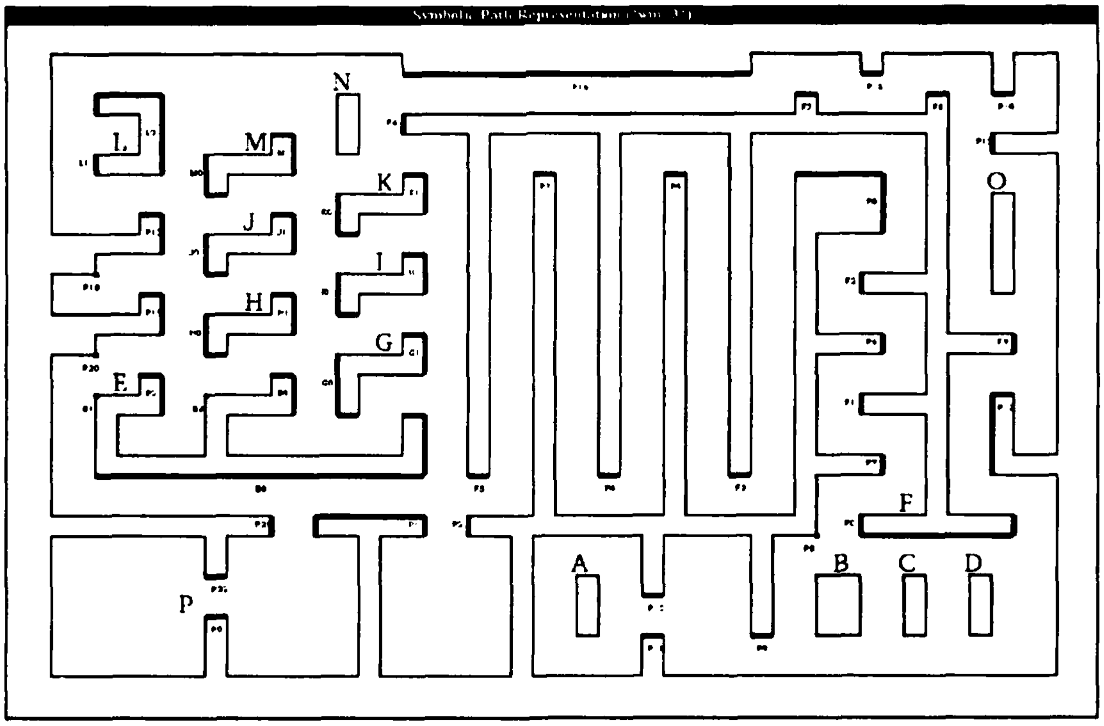
```
(defvar *obstacle-names* (list "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"))

(defvar *polygon-list* '(((520  80) (540  80) (540 140) (520 140))  ;A
                         ((740  80) (780  80) (780 140) (740 140))  ;B
                         ((820  80) (840  80) (840 140) (820 140))  ;C
                         ((880  80) (900  80) (900 140) (880 140))  ;D
                         (( 80 240) (380 240) (380 300) (360 300)   ;E
                          (360 260) (200 260) (200 300) (260 300)
                          (260 340) (240 340) (240 320) (180 320)
                          (180 260) (100 260) (100 300) (140 300)
                          (140 340) (120 340) (120 320) ( 80 320))
                         ((780 180) (920 180) (920 200) (860 200)   ;F
                          (860 360) (920 360) (920 380) (860 380)
                          (860 620) (840 620) (840 600) (740 600)
                          (740 620) (720 620) (720 600) (360 600)
                          (360 580) (420 580) (420 240) (440 240)
                          (440 580) (540 580) (540 240) (560 240)
                          (560 580) (660 580) (660 240) (680 240)
                          (680 580) (840 580) (840 440) (780 440)
                          (780 420) (840 420) (840 320) (780 320)
                          (780 300) (840 300) (840 200) (780 200))
                         ((300 300) (320 300) (320 340) (380 340)   ;G
                          (380 380) (360 380) (360 360) (300 360))
                         ((180 360) (200 360) (200 380) (260 380)   ;H
                          (260 420) (240 420) (240 400) (180 400))
                         ((300 400) (320 400) (320 420) (380 420)   ;I
                          (380 460) (360 460) (360 440) (300 440))
                         ((180 440) (200 440) (200 460) (260 460)   ;J
                          (260 500) (240 500) (240 480) (180 480))
                         ((300 480) (320 480) (320 500) (380 500)   ;K
                          (380 540) (360 540) (360 520) (300 520))
                         (( 80 540) (140 540) (140 620) ( 80 620)   ;L
                          ( 80 600) (120 600) (120 560) ( 80 560))
                         ((180 520) (200 520) (200 540) (260 540)   ;M
 (260 580) (240 580) (240 560) (180 560))
((300 560) (320 560) (320 620) (300 620))        ;N
((900 420) (920 420) (920 520) (900 520))        ;O
(( 40  40) ( 40 180) (180 180) (180 140)         ;P
 (200 140) (200 180) (240 180) (240 200)
 ( 40 200) ( 40 360) ( 80 360) ( 80 380)
 (140 380) (140 420) (120 420) (120 400)
 ( 40 400) ( 40 440) ( 80 440) ( 80 460)
 (140 460) (140 500) (120 500) (120 480)
 ( 40 480) ( 40 660) (360 660) (360 640)
 (680 640) (680 660) (780 660) (780 640)
 (800 640) (800 660) (900 660) (900 620)
 (920 620) (920 660) (960 660) (960 580)
 (900 580) (900 560) (960 560) (960 260)
 (920 260) (920 320) (900 320) (900 240)
 (960 240) (960  40) (600  40) (600  80)
 (580  80) (580  40) (480  40) (480 180)
 (580 180) (580 120) (600 120) (600 180)
 (680 180) (680  80) (700  80) (700 180)
 (740 180) (740 240) (800 240) (800 260)
 (740 260) (740 360) (800 360) (800 380)
 (740 380) (740 480) (800 480) (800 540)
 (720 540) (720 200) (620 200) (620 540)
 (600 540) (600 200) (500 200) (500 540)
 (480 540) (480 200) (420 200) (420 180)
 (460 180) (460  40) (340  40) (340 180)
 (380 180) (380 200) (280 200) (280 180)
 (320 180) (320  40) (200  40) (200 100)
 (180 100) (180  40))))
```



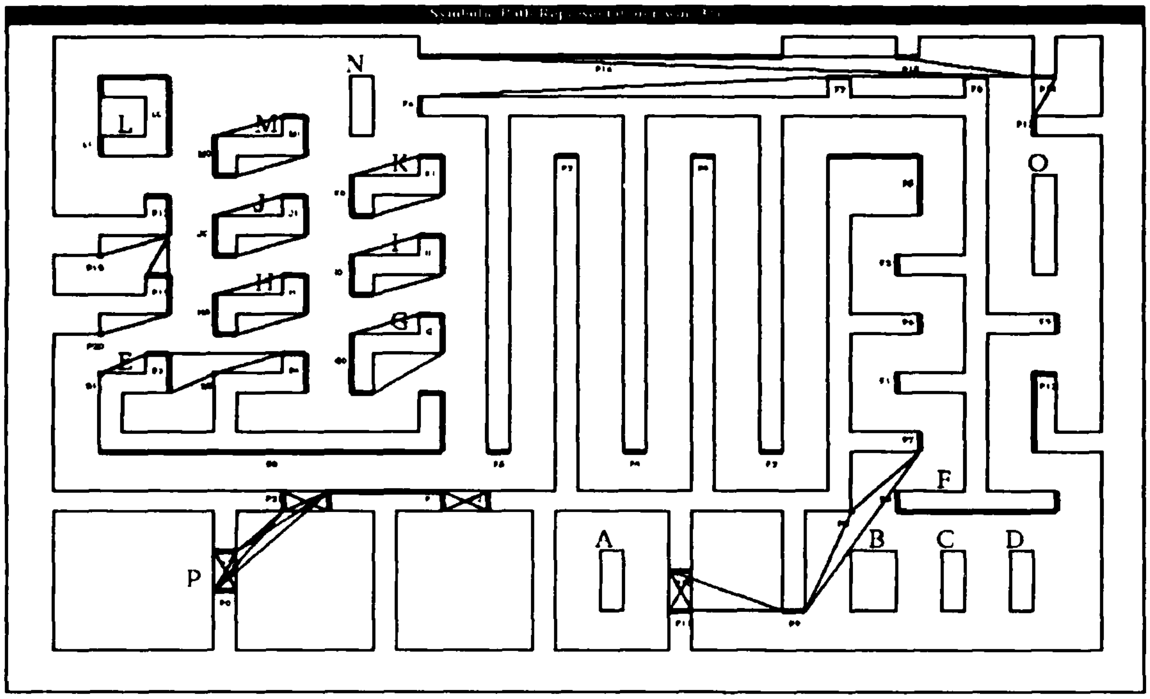
Screen Image of Example World 2 Free and Filled Space



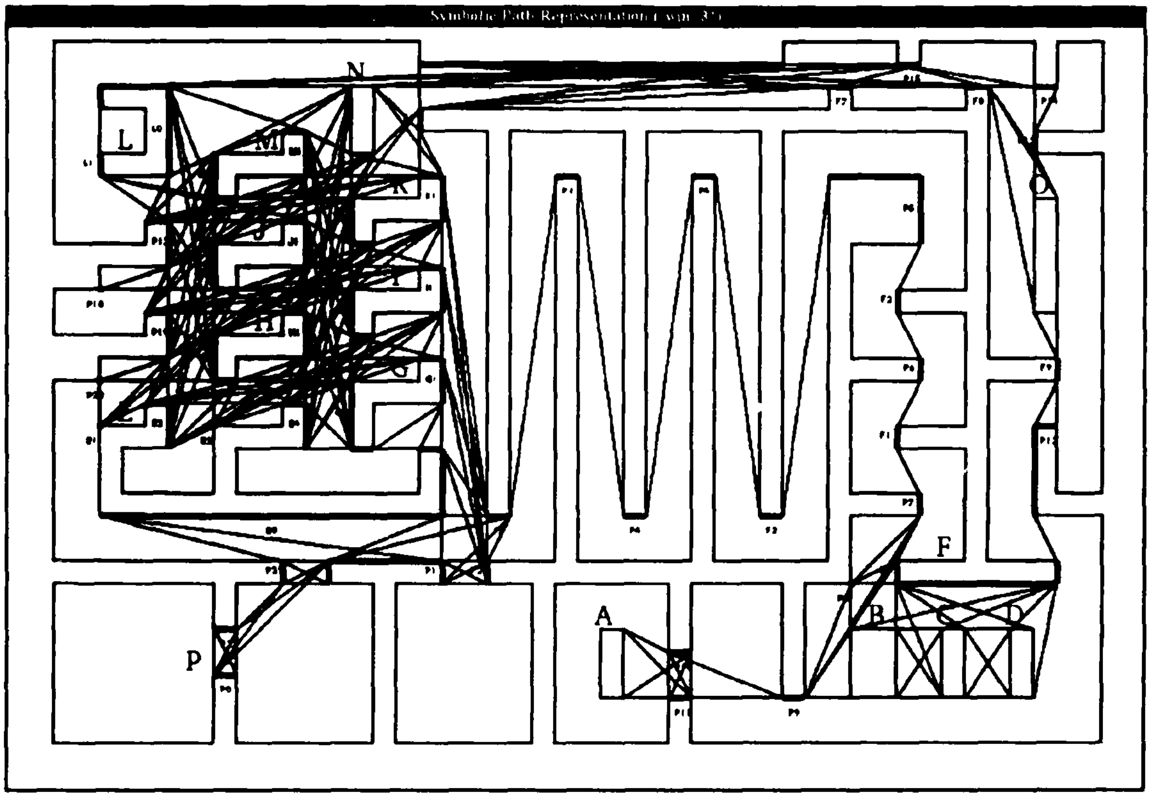
Screen Image of Example World 2 Before Partitioning into Convex Subpolygons



Screen Image of Example World 2 After Partitioning into Convex Subpolygons



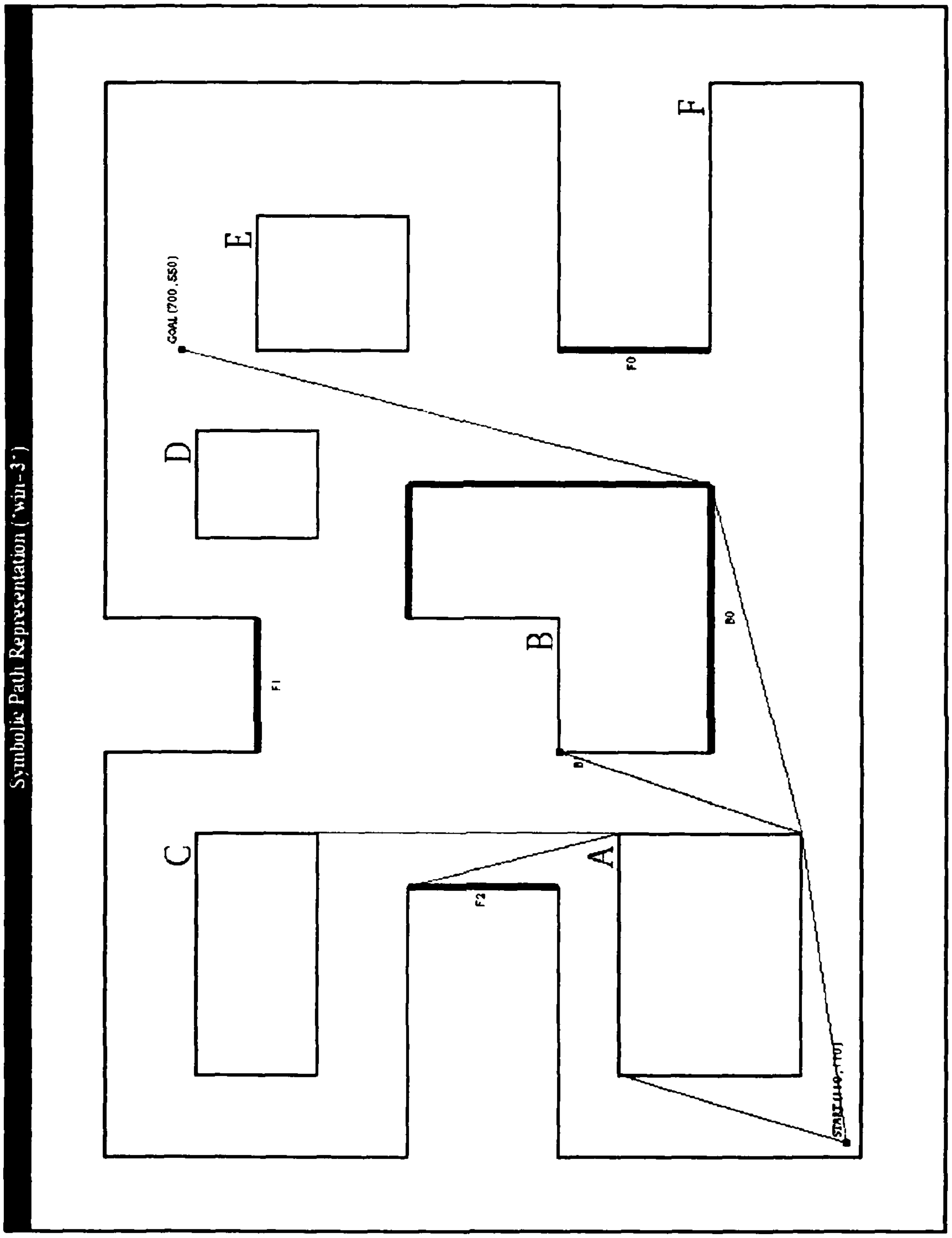
Screen Image of Example World 2 After Construction of Self-Tangents



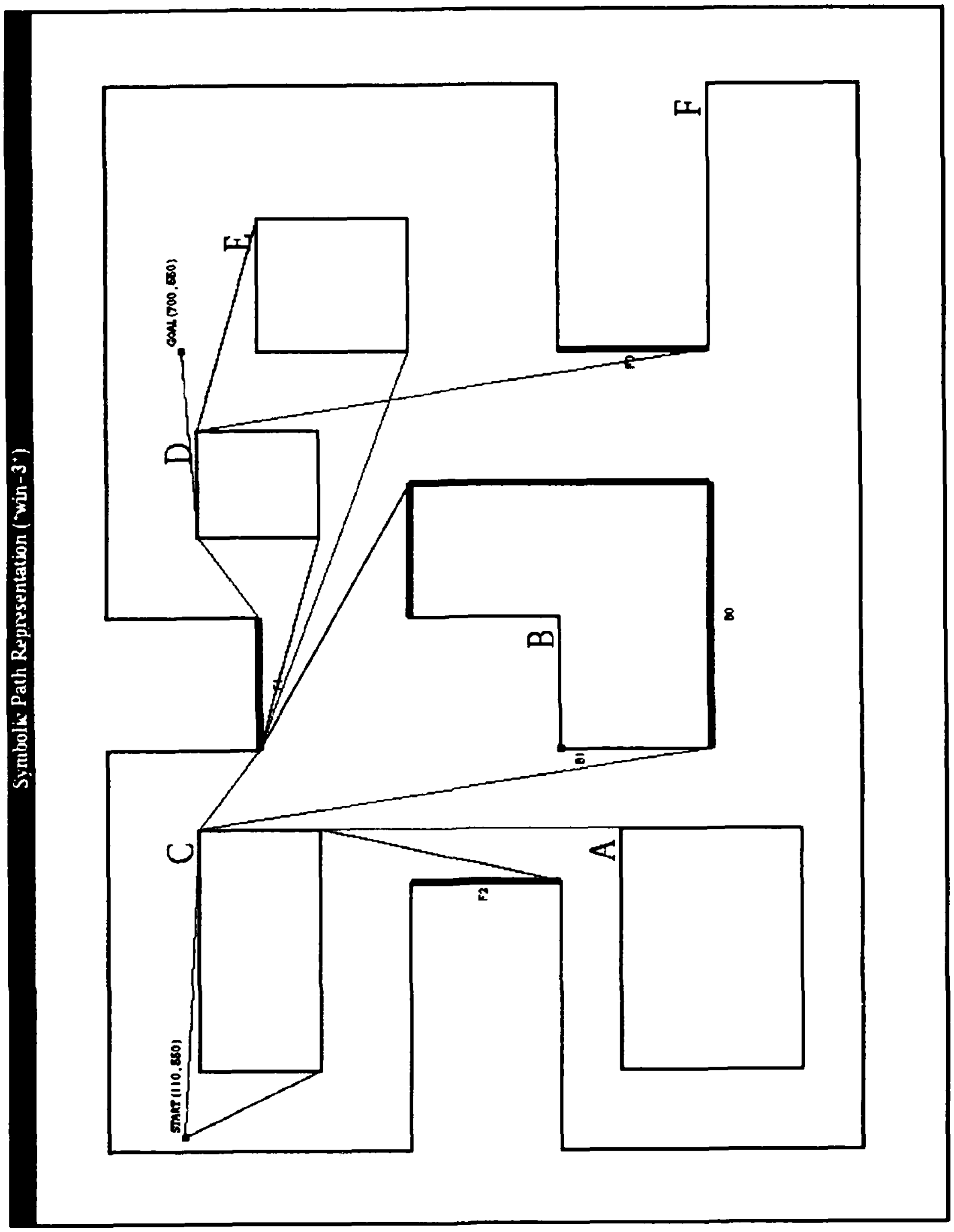
Screen Image of Example World 2 After Construction of All Tangents

APPENDIX C

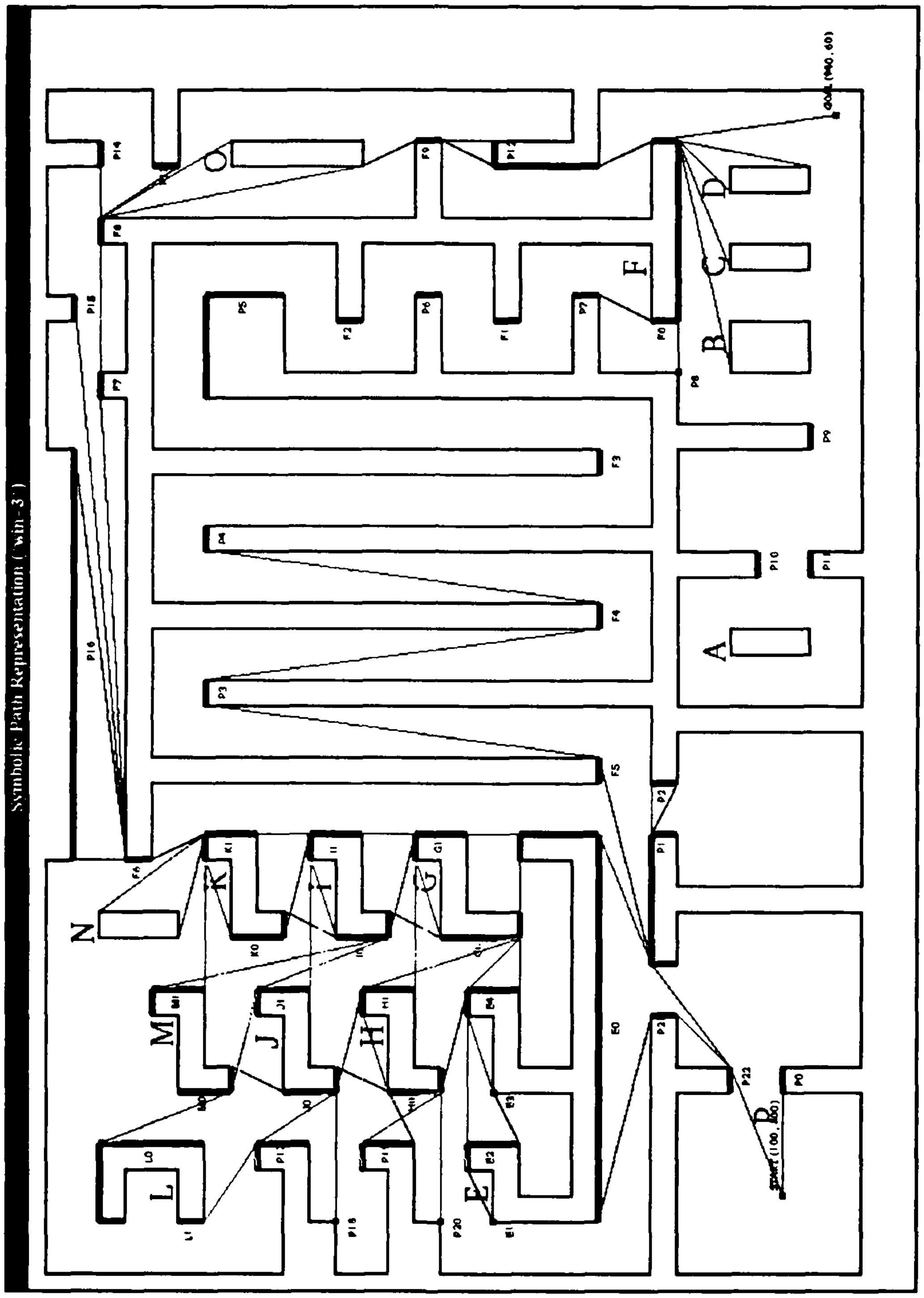
PATH PLANNING RESULTS



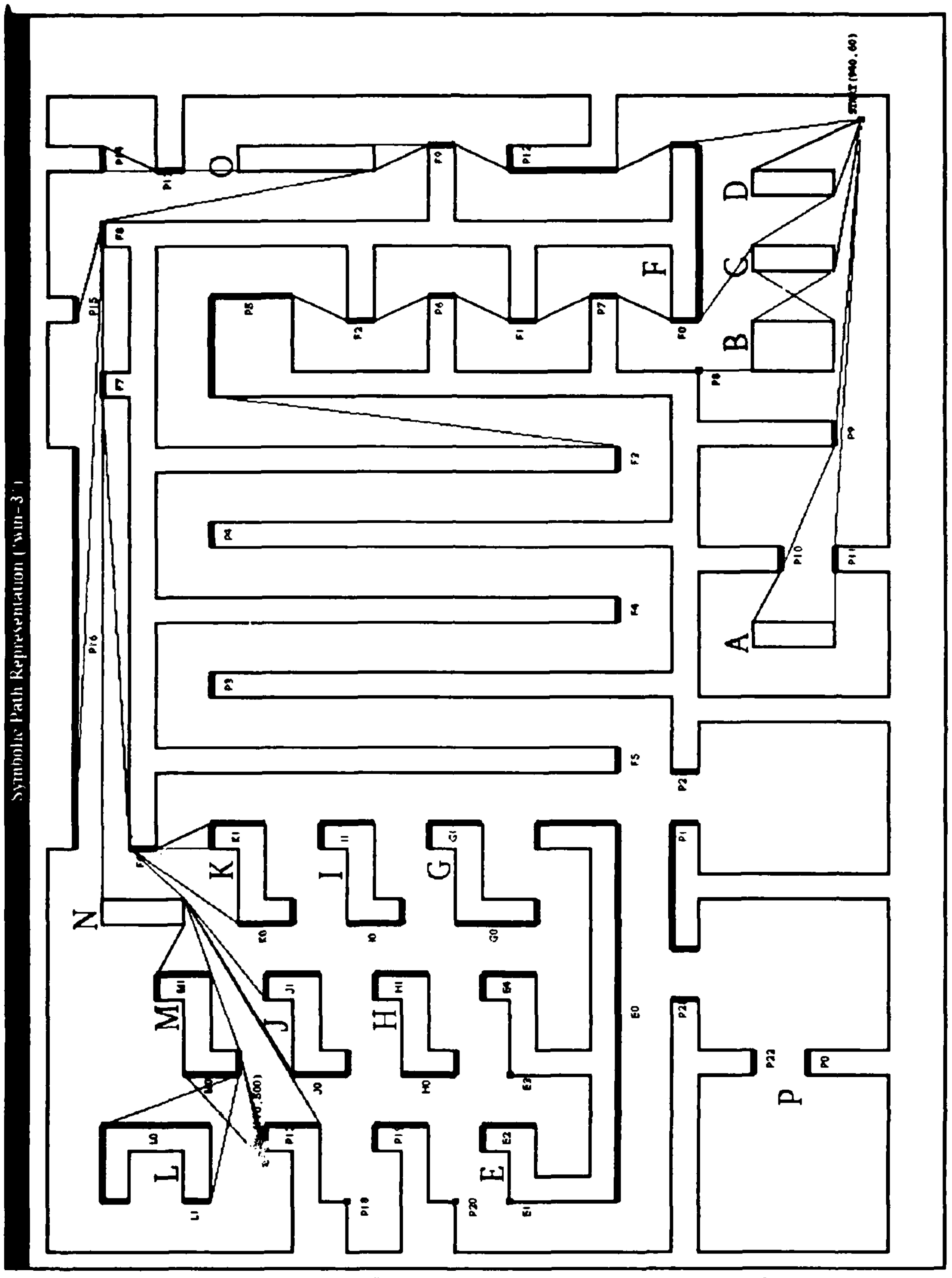
Screen Image of path finding on Example World 1 showing the shortest path expansions from the starting point (110, 110) to the goal (700, 550) along the symbolic path A+B+.

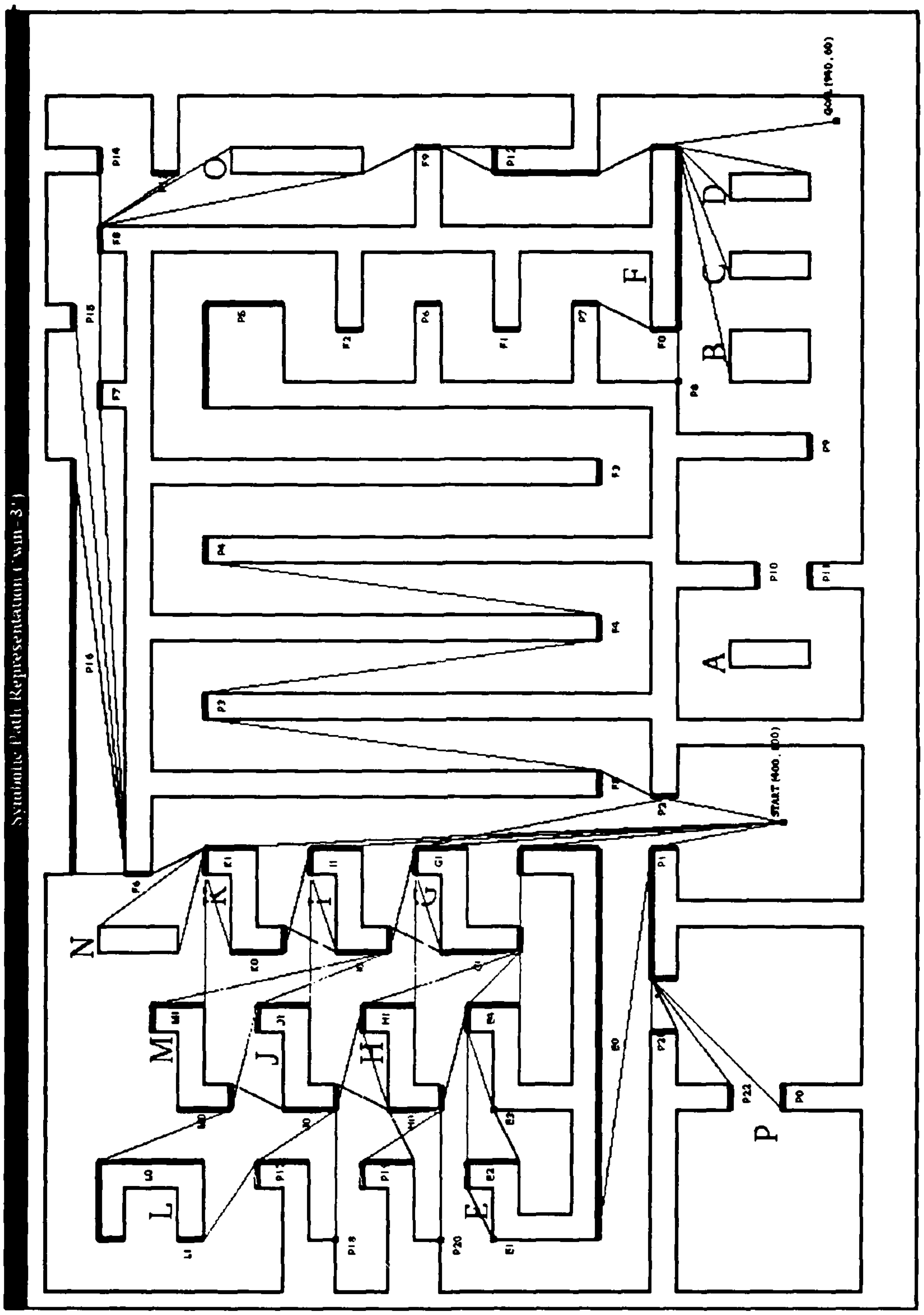


Screen Image of path finding on Example World 1 showing the shortest path expansions from the starting point (110, 550) to the goal (700, 550) along the symbolic path C-F1+D-.

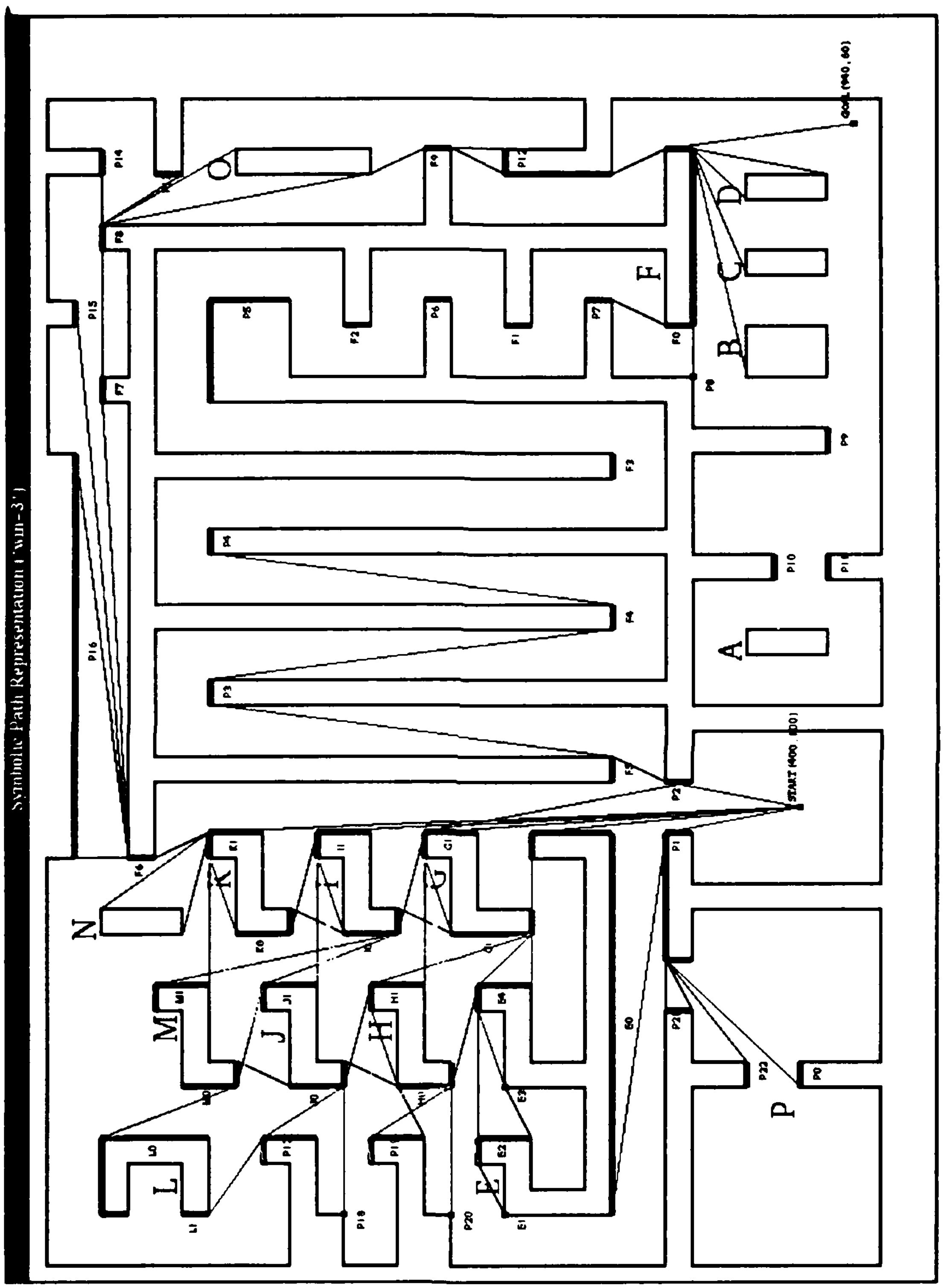


Screen Image of path finding on Example World 2 showing the shortest path expansions from the starting point (100, 100) to the goal (940, 60).


Screen Image of path finding on Example World 2 showing the shortest path expansions from the starting point (940, 60) to the goal (110, 500).



Screen Image of path finding on Example World 2 showing the shortest path expansions from the starting point (110, 110) to the goal (110, 500).



Screen Image of path finding on Example World 2 showing the shortest path expansions from the starting point (400, 100) to the goal (940, 60).

APPENDIX D

PROGRAM LISTING

```
;;;-----------------------------------------------
;;;
;;;      FILENAME:    SETUP.LISP
;;;      AUTHOR:      JERRY A. CRANE
;;;      DATE:        14 AUG 1991
;;;
;;;      DESCRIPTION:
;;; Contains the initialization actions and file loading
;;; to support the common tangent path planner.
;;; SETUP runs the complete path planner, while
;;;  SETUP-1 processes the world and stores the
;;;  results in an executeable file by using the
;;;  LISP dumplisp macro.
;;;
;;;-----------------------------------------------

;***********************************************
;
; BATCH ACTIONS TAKEN TO SETUP DATA STRUCTURES AND
; INITIALIZE POINTERS AND SLOT VALUES ARE CONTAINED
; IN "SETUP". ALL METHODS AND FUNCTIONS ARE ORGANIC
; TO WORLD-DEF.LISP EXCEPT "POINT-POSITION" WHICH
; IS IN "TANGENT-FUNCTIONS.LISP".
;***********************************************
;

;-----------------------------------------------
;    DEFUN SETUP
;-----------------------------------------------
(defun setup ()
  (load "world-def.lisp")
 (let ((answer "no"))
   (load "tangent-functions.lisp")
   (load "concave.lisp")
   (format t "DISPLAY WITH X-WINDOWS? (YES/NO) ")
   (setf answer (read))
   (format t "ANSWER = ( ~a )" answer)
   (cond ((equalp answer 'yes)
        (setf *display* "yes")
        (load "new-window.lisp"))
       (t (setf *display* "no")))
(format t "ENTER OBSTACLE FILE NAME:  ")
(setf answer (read))
;  (setf answer (read))
(format t "~%~%OBSTACLE FILE TO PROCESS = ( ~a )~%~%" answer)
(load answer)
  ;     INITIALIZE NEW VARIABLE
(setf *my-list* nil)
;     CONVERT EACH SET OF (X,Y) COORDINATES
;     INTO VERTEX DATA STRUCTURE
(setf *my-list* (convert-polygon-point-list *polygon-list*))
;     CONNECT PREDECESSOR AND SUCCESSOR
;     POINTERS OF ALL VERTICES
(setf *my-list* (coordinate-conversion *my-list*))
;     CONVERT LIST OF VERTICE DATA STRUCTURES
;     INTO POLYGON DATA STRUCTUREs
(setf *my-list* (polygon-conversion *my-list* *polygon-list*))
; (load "tangent-functions.lisp")
;     CLASSIFY EACH VERTEX AS TO INTERIOR OR
;     EXTERIOR, AND EACH POLYGON AS EITHER
;     CONVEX OR CONCAVE.
(determine-vertex-types *my-list*)
;     LOAD THE SLOT EXTERIOR-VERTICE-LIST
;     FOR CONCAVE POLYGONS WITH THE VERTICES
;     WHICH WILL HAVE TANGENTS.  ALL INTERIOR
;     VERTICES ARE OMITTED TO REDUCE PROCESSING
;     AND IDENTIFICATION TIME LATER.
;     PROCESS ALL CONCAVE POLYGONS INTO CONVEX
;     SUB-POLYGONS.  EACH SUB-POLYGON WILL BE
;     USED DURING SYMBOLIC PATH EVALUATION
(create-convex-sub-polygons *my-list*)
(if (equal *display* "yes")
   (display-convex-sub-polygons *my-list*))
(if (equal *display* "yes")
   (display-polygons *my-list*))
(dolist (poly *my-list*)
  (if (equal (type poly) "concave")
     (find-self-tangents poly *my-list*)))
(locate-some-tangents (first *my-list*) (rest *my-list*) *my-list*)
(collect-tangents *my-list*)
(load "new-path.lisp")
(get-start-and-goal-coordinates-2)
(build-tangents *start-point* *my-list*)
(sort-and-expand-best-path)))

;--------------------------------------------------
```

```
;    DEFUN SETUP-1
;--------------------------------------------------
(defun setup-1 ()
  (load "world-def.lisp")
 (let ((answer "no"))
   (load "tangent-functions.lisp")
   (load "concave.lisp")
   (setf *display* "no")
   (format t "ENTER OBSTACLE FILE NAME:  ")
   (setf answer "obstacle.lisp")
   (format t "~%~%OBSTACLE FILE TO PROCESS = ( ~a )" answer)
   (load answer)
   (setf *my-list* nil)
   (setf *my-list* (convert-polygon-point-list *polygon-list*))
   (setf *my-list* (coordinate-conversion *my-list*))
   (setf *my-list* (polygon-conversion *my-list* *polygon-list*))
   (determine-vertex-types *my-list*)
   (create-convex-sub-polygons *my-list*)
   (dolist (poly *my-list*)
     (if (equal (type poly) "concave")
        (find-self-tangents poly *my-list*)))
   (locate-some-tangents (first *my-list*) (rest *my-list*) *my-list*)
   (collect-tangents *my-list*)
   (dumplisp :name "map1" :read-init-file t)))
```

```
;;;---------------------------------------------------
;;;
;;;      FILENAME:     WORLD-DEF.LISP
;;;      AUTHOR:       JERRY A. CRANE
;;;      DATE:         14 AUG 1991
;;;      DESCRIPTION:
;;; Contains the data structures and initialization
;;; actions taken to convert polygons consisting of
;;; coordinate point lists into CLOS objects.
;;;
;;;---------------------------------------------------

(setf *print-circle* 1)
(defvar *display* "yes")
(defvar *my-list* nil)

(setf *my-list* nil)

;-------------------------------------------------
;    DEFCLASS POINT
;-------------------------------------------------
(defclass point ()
    ((x-coord        :initarg :x-coord         :accessor  x-coord)
     (y-coord        :initarg :y-coord         :accessor  y-coord)
     (name           :initarg :name            :accessor  name)))

;-------------------------------------------------
;    DEFCLASS VERTEX
;-------------------------------------------------
(defclass vertex ()
    ((coordinates    :initarg :coordinates     :accessor  coordinates)
     (vertex-type    :initarg :vertex-type     :accessor  vertex-type)
     (predecessor    :initarg :predecessor     :accessor  predecessor)
     (successor      :initarg :successor       :accessor  successor)
     (plus-tangents  :initarg :plus-tangents   :accessor  plus-tangents)
     (minus-tangents :initarg :minus-tangents :accessor  minus-tangents)
     (parent-name    :initarg :parent-name     :accessor  parent-name)
     (parent         :initarg :parent          :accessor  parent)
     (sub-polygon    :initarg :sub-polygon     :accessor  sub-polygon)
     (vertex-number  :initarg :vertex-number   :accessor  vertex-number)))

;-------------------------------------------------
;    DEFCLASS POLYGON
;-------------------------------------------------
(defclass polygon ()
    ((name               :initarg   :name             :accessor  name)
     (type-of-polygon    :initarg   :type             :accessor  type)
```

```
      (vertice-list          :initarg   :vertice-list         :accessor  vertice-list)
      (exterior-vertice-list :initform  ()                    :accessor  exterior-vertice-list)
      (number-of-vertices    :initarg   :number-of-vertices   :accessor number-of-vertices)
      (xy-coordinates        :initarg   :xy-coordinates       :accessor  xy-coordinates)
      (plus-tangents         :initarg   :plus-tangents        :accessor  plus-tangents)
      (minus-tangents        :initarg   :minus-tangents       :accessor  minus-tangents)
      (convex-sub-polygons   :initarg   :sub-polygons         :accessor  sub-polygons)
      (plus-mode-path        :initarg   :plus-mode            :accessor  plus-mode)
      (minus-mode-path       :initarg   :minus-mode           :accessor  minus-mode)))

;--------------------------------------------------
;    DEFCLASS CONVEX-SUB-POLYGON
;--------------------------------------------------
(defclass convex-sub-polygon ()
     ((name                  :initarg   :name                 :accessor  name)
      (vertice-list          :initarg   :vertice-list         :accessor  vertice-list)
      (plus-tangents         :initarg   :plus-tangents        :accessor  plus-tangents)
      (minus-tangents        :initarg   :minus-tangents       :accessor  minus-tangents)
      (plus-mode-path        :initarg   :plus-mode            :accessor  plus-mode)
      (minus-mode-path       :initarg   :minus-mode           :accessor  minus-mode)))

;--------------------------------------------------
;    DEFCLASS PATH-NODE
;--------------------------------------------------
(defclass path-node ()
     ((path-mode          :initarg   :path-mode         :accessor   path-mode)
      (landing-vertex     :initarg   :landing-vertex    :accessor   landing-vertex)
      (from-vertex        :initarg   :from-vertex       :accessor   from-vertex)
      (from-polygon       :initarg   :from-polygon      :accessor   from-polygon)
      (from-mode          :initarg   :from-mode         :accessor   from-mode)
      (cost               :initarg   :cost              :accessor   cost)
      (symbolic-path      :initform  nil                :accessor   path)
      (path-area          :initarg   :path-area         :accessor   path-area)
      (distance-to-goal   :initarg   :distance          :accessor   distance)
      (total-path-cost    :initarg   :total-path-cost   :accessor   total-path-cost)))

;--------------------------------------------------
;    DEFUN MAKE-PATH_NODE  (POLYGON)
;--------------------------------------------------
(defmethod make-path-node    ((landing-vertex vertex)
                              (from-vertex    vertex)
                              (from-polygon   polygon)
                               from-mode
                               cost)
     (make-instance 'path-node
                    :landing-vertex  landing-vertex
                    :from-vertex     from-vertex
```

```
                            :from-polygon   from-polygon
                            :from-mode      from-mode
                            :cost           cost))

;-------------------------------------------------
;    DEFUN MAKE-PATH_NODE  (CONVEX-SUB-POLYGON)
;-------------------------------------------------
(defmethod make-path-node    ((landing-vertex vertex)
                              (from-vertex    vertex)
                              (from-polygon   convex-sub-polygon)
                               from-mode
                               cost)

 (make-instance 'path-node
                   :landing-vertex landing-vertex
                   :from-vertex    from-vertex
                   :from-polygon   from-polygon
                   :from-mode      from-mode
                   :cost           cost))

;-------------------------------------------------
;    DEFCLASS TANGENT-LINE
;-------------------------------------------------
(defclass tangent-line ()
     ((end-point-1    :accessor end-point-1     :initarg :end-point-1)
      (end-point-2    :accessor end-point-2     :initarg :end-point-2)
     (tangent-type    :accessor type            :initarg :type)
     (distance        :accessor distance        :initarg :distance)
     (angle           :accessor angle           :initarg :angle)))

;-------------------------------------------------
;    DEFUN MAKE-POINT
;-------------------------------------------------
(defun make-point  (x-coordinate
                    y-coordinate
                    &optional name)

   (make-instance 'point
                   :x-coord x-coordinate
                   :y-coord y-coordinate
                   :name    name))

;-------------------------------------------------
;    DEFUN MAKE-VERTEX
;-------------------------------------------------
(defun make-vertex (x-coordinate
                    y-coordinate
```

```
                  &optional name)

   (make-instance 'vertex
                  :coordinates    (make-point x-coordinate y-coordinate name)
                  :vertex-type    "exterior"
                  :plus-tangents  nil
                  :minus-tangents nil
                  :parent-name    nil
                  :sub-polygon    nil
                  :vertex-number  nil))

;-----------------------------------------------
;   DEFMETHOD MAKE-TANGENT-LINE
;-----------------------------------------------
(defmethod make-tangent-line ((first-point vertex)
                              (second-point vertex)
                              &optional tangent-mode)

   (make-instance 'tangent-line
                  :end-point-1  first-point
                  :end-point-2  second-point
                  :type         tangent-mode))

;-----------------------------------------------
;     DEFMETHOD STRAIGHT-LINE-DISTANCE
;-----------------------------------------------
; Determines the straight line distance
; between the two points.
;-----------------------------------------------
(defmethod straight-line-distance   ((first-point vertex)
                                     (second-point vertex))

   (let ((x1 (x-coord (coordinates first-point)))
         (y1 (y-coord (coordinates first-point)))
         (x2 (x-coord (coordinates second-point)))
         (y2 (y-coord (coordinates second-point))))
   (sqrt (+ (expt (- x1 x2) 2)
            (expt (- y1 y2) 2)))))

;-----------------------------------------------
;   DEFUN RADIANS-TO-DEGREES
;-----------------------------------------------
(defun radians-to-degrees (angle)
      (/ (* angle 360) (* 2 pi)))

;-----------------------------------------------
;   DEFUN BETA1
```

```
;-------------------------------------------------
(defmethod beta1    ((first-point vertex)
                     (second-point vertex))

    (let ((x1 (x-coord (coordinates first-point)))
          (y1 (y-coord (coordinates first-point)))
          (x2 (x-coord (coordinates second-point)))
          (y2 (y-coord (coordinates second-point))))
    (/ (* (atan (- y2 y1) (- x2 x1)) 360)
          (* 2 pi))))

;-------------------------------------------------
;    DEFUN CONVERT-POLYGON-COORDINATES-TO-VERTICES
;-------------------------------------------------
(defun convert-polygon-coordinates-to-vertices (coordinate-list)

    (let ((list-length (length coordinate-list)))
          (cond  ((<= list-length 1)
                        (list (make-vertex (first (first coordinate-list)) (second (first coordinate-list)))))
                     (t
                        (cons (make-vertex (first (first coordinate-list)) (second (first coordinate-list)))
          (convert-polygon-coordinates-to-vertices (rest coordinate-list)))))))

;-------------------------------------------------
;    DEFUN CONVERT-POLYGON-POINT-LIST
;-------------------------------------------------
(defun convert-polygon-point-list (polygon-list)

    (let ((list-length (length polygon-list)))
          (cond ((<= list-length 1)
                  (list (convert-polygon-coordinates-to-vertices (first polygon-list))))
           (t
                  (cons (convert-polygon-coordinates-to-vertices (first polygon-list))
    (convert-polygon-point-list (rest polygon-list)))))))

;-----------------------------------------------
;    DEFUN COORDINATE-CONVERSION
;-----------------------------------------------
(defun coordinate-conversion (polygon-vertice-list)

    (dolist (polygon polygon-vertice-list polygon-vertice-list)
          (link-polygon-vertices polygon)))

;-----------------------------------------------
;    DEFUN MAKE-POLYGON
;----------------------------------------------
(defun make-polygon    (name
                        type
                        vertice-list
                        number-of-vertices
                        xy-coordinates)

    (make-instance 'polygon
                     :name name
                     :type type
                     :vertice-list vertice-list
                     :number-of-vertices number-of-vertices
                     :xy-coordinates xy-coordinates))

;------------------------------------------------
;   DEFUN POLYGON-CONVERSION
;------------------------------------------------
(defun polygon-conversion (polygon-vertice-list
                           coordinate-list)

 (let ((temp-list ())
      (temp-name (first *obstacle-names*)))
  (setf *obstacle-names* (rest *obstacle-names*))
  (cond ((null (second polygon-vertice-list))
        (list (make-polygon temp-name
                            "convex"
                            (first polygon-vertice-list)
                            (length (first polygon-vertice-list))
                            (dolist (vertex (first coordinate-list) temp-list)
                               (setf temp-list
                                 (cons (first vertex)
                                       (cons (second vertex)
                                          temp-list)))))))
       (t (cons (make-polygon temp-name
                              "convex"
                              (first polygon-vertice-list)
                              (length (first polygon-vertice-list))
                              (dolist (vertex (first coordinate-list) temp-list)
                               (setf temp-list
                                 (cons (first vertex)
                                       (cons (second vertex)
                                            temp-list)))))
           (polygon-conversion (rest polygon-vertice-list)
                               (rest coordinate-list)))))))

;------------------------------------------------
;   DEFMETHOD DETERMINE-OBSTACLE-VERTEX-TYPE
;------------------------------------------------
(defmethod determine-obstacle-vertex-type ((current-vertex vertex))
(let* ((current-point  (coordinates current-vertex))
      (next-point    (coordinates (successor current-vertex)))
      (previous-point (coordinates (predecessor current-vertex)))
      (result (point-position next-point
                             previous-point
                             current-point)))
  (cond ((> result 0)
        (setf (vertex-type current-vertex) "exterior"))
       ((< result 0)
        (setf (vertex-type current-vertex) "interior")))))

;-------------------------------------------------
;    DEFUN DETERMINE-VERTEX-TYPES
;-------------------------------------------------
(defun determine-vertex-types (polygon-list)
 (dolist (poly polygon-list)
   (dolist (point (vertice-list poly))
        (setf (parent-name point) (name poly))
        (setf (parent point) poly)
        (determine-obstacle-vertex-type point)
        (cond ((equal (vertex-type point) "interior")
              (setf (type poly) "concave"))
             (t (setf (exterior-vertice-list poly)
                    (cons point (exterior-vertice-list poly))))))))

;-------------------------------------------------
;    DEFUN LINK-POLYGON-VERTICES
;-------------------------------------------------
(defun link-polygon-vertices (polygon-vertice-list)
 (let ((list-length (length polygon-vertice-list)))
  (dotimes (list-index list-length polygon-vertice-list)
   (cond ((equal list-index 0)                    ; first vertex
        (setf (predecessor (first polygon-vertice-list))
          (first (last polygon-vertice-list)))
        (setf (successor (first polygon-vertice-list))
          (second polygon-vertice-list)))
       ((equal list-index (- list-length 1))          ; last vertex
        (setf (successor (first (last polygon-vertice-list)))
          (first polygon-vertice-list))
        (setf (predecessor (first (last polygon-vertice-list)))
          (nth (- list-index 1) polygon-vertice-list)))
       (t (connect-links (nth (- list-index 1)
                         polygon-vertice-list)       ; all others
                    (nth list-index polygon-vertice-list)
                    (nth (+ list-index 1) polygon-vertice-list)))))))
```

```
;---------------------------------------------------------
;    DEFUN CONNECT-LINKS
;---------------------------------------------------------
(defun connect-links (first-point point-to-process second-point)
 (setf (predecessor point-to-process) first-point)
 (setf (successor point-to-process) second-point))

;--------------------------------------------------------------
;    DEFUN SET-EXTERIOR-VERTICE-LIST
;--------------------------------------------------------------
(defun set-exterior-vertice-list (polygon-list)
 (dolist (polygon polygon-list)
  (if (equal (type polygon) "concave")
     (dolist (current-vertex (reverse (vertice-list polygon)))
       (if (equal (vertex-type current-vertex) "exterior")
          (setf (exterior-vertice-list polygon)
           (cons current-vertex
                (exterior-vertice-list polygon))))))))
```

```
;;;-----------------------------------------------
;;;
;;;     FILENAME:     TANGENT-FUNCTIONS.LISP
;;;     AUTHOR:       JERRY A. CRANE
;;;     DATE:         14 AUG 1991
;;;     DESCRIPTION:
;;;  CONTAINS THE FUNCTIONS TO DETERMINE LINE
;;;  INTERSECTION FOR A GIVEN LIST OF POLYGONS
;;;  AND A LINE.  ALSO CONTAINS THE FUNCTIONS
;;;  NECESSARY TO FIND POINT-TO-OBSTACLE
;;;  TANGENT LINES, AS WELL AS COMMON-TANGENTS
;;;  FOR TWO GIVEN OBSTACLES.
;;;
;;;-----------------------------------------------

(defvar *TEST* "false")

;-----------------------------------------------------
;              POINT-POSITION
;-----------------------------------------------------
;  DETERMINE THE RELATIVE POSITION OF THE TEST
;  POINT TO THE LINE FROM POINT-1 TO POINT-2
;  USING THE FORMULA
;    (((X1 - X) * (Y2 - Y)) - ((Y1 - Y) * (X2 - X)))
;-----------------------------------------------------

(defmethod point-position ((test-point point)
                           (point-1   point)
                           (point-2   point))

 (let ((x  (x-coord test-point))
       (y  (y-coord test-point))
       (x1 (x-coord point-1))
       (y1 (y-coord point-1))
       (x2 (x-coord point-2))
       (y2 (y-coord point-2)))

   (- (* (- x1 x) (- y2 y))
      (* (- y1 y) (- x2 x)))))

;-----------------------------------------------------
;   LINE-INTERSECTION  (TANGENT-LINE & 2 - POINTS)
;-----------------------------------------------------
;   DETERMINE IF THE TANGENT LINE AND THE LINE
;   FORMED BY THE FIRST AND SECOND VERTICES
;   INTERSECT.  RETURNS "nil" or "intersection".
;-----------------------------------------------------
```

```
(defmethod intersection-test ((line        tangent-line)
                              (first-vertex  vertex)
                              (second-vertex vertex))
 (let* ((starting-point    (coordinates (end-point-1 line)))
        (end-point          (coordinates (end-point-2 line)))
        (first-point       (coordinates first-vertex))
        (second-point       (coordinates second-vertex))
     (first-point-sign   (point-position first-point
                                  starting-point
                                  end-point))
     (second-point-sign  (point-position second-point
                                  starting-point
                                  end-point))
     (starting-point-sign (point-position starting-point
                                  first-point
                                  second-point))
     (end-point-sign     (point-position end-point
                                  first-point
                                  second-point)))
  (cond ((or (equal first-point starting-point)
          (equal first-point end-point)            ; ONE OF THE POINTS TO TEST
          (equal second-point starting-point)        ; IS ONE OF THE END POINTS OF
          (equal second-point end-point)) nil)       ; THE LINE BEING TESTED
      ((and (> first-point-sign  0)
          (> second-point-sign 0)) nil)             ; BOTH GREATER THAN ZERO
      ((and (< first-point-sign  0)
          (< second-point-sign 0)) nil)             ; BOTH LESS THAN ZERO
        ((and (= first-point-sign 0)
              (= second-point-sign 0)
              (= starting-point-sign 0)
              (= end-point-sign 0))
         (cond ((or (and (< (x-coord first-point)
                            (x-coord starting-point))
                         (< (x-coord second-point)
                            (x-coord starting-point)))
                    (and (> (x-coord first-point)
                            (x-coord starting-point))
                         (> (x-coord second-point)
                            (x-coord starting-point)))
                    (and (< (y-coord first-point)
                            (y-coord starting-point))
                         (< (y-coord second-point)
                            (y-coord starting-point)))
                    (and (> (y-coord first-point)
                            (y-coord starting-point))
                         (> (y-coord second-point)
                            (y-coord starting-point)))) nil)))
          ((and (and (>= first-point-sign   0)
                     (<= second-point-sign  0))    ; LINE FORMED BY THE TWO
             (and (<= starting-point-sign 0)      ; TEST POINTS INTERSECTS, HAS
                (>= end-point-sign     0)))   ; ONE OR BOTH POINTS ON THE
        "intersection")                    ; TANGENT LINE
        ((and (and (<= first-point-sign   0)
                (>= second-point-sign  0))    ; LINE FORMED BY THE TWO
             (and (>= starting-point-sign 0)      ; TEST POINTS INTERSECTS, HAS
                (<= end-point-sign     0)))   ; ONE OR BOTH POINTS ON THE
        "intersection")                       ; TANGENT LINE
          (t nil))))

;-------------------------------------------------------
;    CHECK-FOR-POLYGON-INTERSECTION
;-------------------------------------------------------
;    DETERMINE IF THE TEST-LINE INTERSECTS A
;    THE GIVEN POLYGON.  RETURNS THE VALUE "nil"
;    OR "intersection."  TESTS THE FIRST VERTEX
;    AND THE PRECEEDING VERTEX FOR THE RESULT.
;-------------------------------------------------------
(defun check-for-polygon-intersection (test-line
                              polygon-vertice-list)
 (cond ((null polygon-vertice-list) nil)
     (t (let* ((initial-polygon-vertex (first polygon-vertice-list))
             (second-polygon-vertex  (predecessor initial-polygon-vertex))
             (test-result          (intersection-test test-line
                                          initial-polygon-vertex
                                          second-polygon-vertex)))
          (cond ((equal test-result "intersection")
               "intersection")
             (t (check-for-polygon-intersection test-line
                                     (rest polygon-vertice-list))))))))
;-------------------------------------------------------
;    CHECK-VISIBILITY
;-------------------------------------------------------
;    TESTS WHETHER THE TEST LINE INTERSECTS
;    ANY OF THE OBSTACLES IN THE PROVIDED
;    OBSTACLE LIST.  RETURNS EITHER THE VALUE
;    "nil" or "intersection."
;-------------------------------------------------------
(defmethod check-visibility ((test-line tangent-line)
                     &optional (obstacle-list
                             *my-list*
                             obstacle-list-supplied-p))
 (cond ((null obstacle-list) nil)
     (t (let ((result (check-for-polygon-intersection
                 test-line
                (vertice-list (first obstacle-list)))))
            (cond ((equal result "intersection")
                   "intersection")
                  (t (check-visibility
                       test-line
                       (rest obstacle-list))))))))

;------------------------------------------------------------
;    CHECK-LINE
;------------------------------------------------------------
;    CHECKS THE RELATIVE POSITION OF THE PREDECESSOR
;    AND SUCCESOR VERTICES OF POLYGON 1 VERTEX
;    TO THE LINE FROM POLYGON 2 VERTEX TO
;    POLYGON 1 VERTEX. RETURNS THE SIGNS OF
;    THE RESULT AS A STRING
;------------------------------------------------------------
(defmethod check-line ((polygon-1-vertex vertex)
                       (polygon-2-vertex vertex))
 (let* ((predecessor-test (point-position (coordinates (predecessor polygon-1-vertex))
                                          (coordinates polygon-1-vertex)
                                          (coordinates polygon-2-vertex)))
        (successor-test  (point-position (coordinates (successor polygon-1-vertex))
                                          (coordinates polygon-1-vertex)
                                          (coordinates polygon-2-vertex))))
  (cond ((and (< predecessor-test 0)
              (> successor-test  0)) "-+")
        ((and (> predecessor-test 0)
              (< successor-test  0)) "+-")
        ((and (> predecessor-test 0)
              (> successor-test  0)) "++")
        ((and (< predecessor-test 0)
              (< successor-test  0)) "--")
        ((and (= predecessor-test 0)
              (> successor-test  0)) "0+")
       ((and (= predecessor-test 0)
              (< successor-test  0)) "0-")
       ((and (> predecessor-test 0)
              (= successor-test  0)) "+0")
       ((and (< predecessor-test 0)
              (= successor-test  0)) "-0"))))
;------------------------------------------------------------
;    FIND-EXTERIOR-VERTEX
;------------------------------------------------------------
(defun find-exterior-vertex (vertex-list)
 (cond ((equal (vertex-type (first vertex-list)) "exterior")
        vertex-list)
       ((null (second vertex-list)) nil)
```

```
      (t (find-exterior-vertex (rest vertex-list)))))

;--------------------------------------------------------
;    ADJACENCY-TEST
;--------------------------------------------------------
(defmethod adjacency-test ((vertex-1 vertex)
                           (vertex-2 vertex))
 (cond ((or (equal (predecessor vertex-1) vertex-2)
            (equal (successor   vertex-1) vertex-2)) "adjacent")
            ((equal vertex-1 vertex-2) "equal")
      (t nil)))

;--------------------------------------------------------
;    FIND-SELF-TANGENTS
;--------------------------------------------------------
(defmethod find-self-tangents ((poly polygon)
                               obstacle-list)
 (locate-self-tangents (first (exterior-vertice-list poly))
                       (rest (exterior-vertice-list poly))
                       obstacle-list))

;--------------------------------------------------------
;    LOCATE-SELF-TANGENTS
;--------------------------------------------------------
(defmethod locate-self-tangents ((fixed-vertex vertex)
                                 exterior-vertice-list
                                 obstacle-list)
 (cond ((null exterior-vertice-list) nil)
       (t
        (dolist (current-vertex exterior-vertice-list)
         (let ((adjacency-test-result (adjacency-test
                                       fixed-vertex
                                       current-vertex)))
          (cond ((equal adjacency-test-result "adjacent") nil)
                (t
                 (let* ((from-fixed-vertex-test (check-line
                                                  fixed-vertex
                                                  current-vertex))
                        (from-current-vertex-test (check-line
                                                   current-vertex
                                                   fixed-vertex))
                        (temp-tangent (make-tangent-line
                                       fixed-vertex
                                       current-vertex)))
                  (cond ((and (equal from-fixed-vertex-test "--")
                              (equal from-current-vertex-test "++"))
                         (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "minus-minus"
                                        obstacle-list))
((and (equal from-fixed-vertex-test  "--")
      (equal from-current-vertex-test "--"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "minus-plus"
                                        obstacle-list))
((and (equal from-fixed-vertex-test  "++")
      (equal from-current-vertex-test "--"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "plus-plus"
                                        obstacle-list))

((and (equal from-fixed-vertex-test  "++")
      (equal from-current-vertex-test "++"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "plus-minus"
                                        obstacle-list))
((and (equal from-fixed-vertex-test  "0+")
      (equal from-current-vertex-test "-0"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "plus-plus"
                                        obstacle-list))
((and (equal from-fixed-vertex-test  "-0")
      (equal from-current-vertex-test "-0"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "minus-plus"
                                        obstacle-list))
((and (equal from-fixed-vertex-test  "-0")
      (equal from-current-vertex-test "0+"))
 (test-verify-and-attach  temp-tangent
                                        fixed-vertex
                                        current-vertex
                                        "minus-minus"
                                        obstacle-list))
```

```
                    ((and (equal from-fixed-vertex-test "+0")
                          (equal from-current-vertex-test "+0"))
                     (test-verify-and-attach temp-tangent
                                              fixed-vertex
                                              current-vertex
                                              "plus-minus"
                                              obstacle-list))
                    (t nil)))))))
       (locate-self-tangents (first exterior-vertice-list)
                              (rest exterior-vertice-list)
                              obstacle-list))))

;---------------------------------------------------------
;   TEST-VERIFY-AND-ATTACH
;---------------------------------------------------------
(defmethod test-verify-and-attach ((temp-tangent tangent-line)
                                   (fixed-vertex vertex)
                                   (current-vertex vertex)
                                   tangent-mode
                                   obstacle-list)
  (let ((intersection-test (verify-tangent-line-visibility
                            temp-tangent
                            obstacle-list)))
    (if (equal intersection-test "valid-tangent-line")
        (attach-tangent temp-tangent
                        fixed-vertex
                        current-vertex
                        tangent-mode))))

;---------------------------------------------------------
;   VERIFY-TANGENT-LINE-VISIBILITY
;---------------------------------------------------------
(defmethod verify-tangent-line-visibility ((line tangent-line)
                                           obstacle-list)
  (let ((visibility-test-result (check-visibility line obstacle-list))
        (point-1 (end-point-1 line))
        (point-2 (end-point-2 line)))
    (cond ((equal visibility-test-result nil)
           (if (equalp *display* "yes")
               (display-tangent-line line))
           "valid-tangent-line")
          (t nil))))

;---------------------------------------------------------
;   ATTACH-TANGENT
;---------------------------------------------------------
(defmethod attach-tangent ((first-tangent-line tangent-line)
                (originating-vertex vertex)
                          (terminating-vertex vertex)
                          first-tangent-mode)
(setf (distance first-tangent-line) (straight-line-distance originating-vertex
                                                              terminating-vertex))
(setf (angle first-tangent-line) (beta1 originating-vertex
                                        terminating-vertex))
(let* ((second-tangent-mode
         (cond ((equal first-tangent-mode "plus-plus")
                "minus-minus")
               ((equal first-tangent-mode "minus-minus")
                "plus-plus")
               (t first-tangent-mode)))
        (second-tangent-line (make-tangent-line
                               terminating-vertex
                               originating-vertex
                               second-tangent-mode)))
  (setf (distance second-tangent-line)
    (straight-line-distance terminating-vertex
                            originating-vertex))
  (setf (angle second-tangent-line)
    (beta1 terminating-vertex
           originating-vertex))
  (setf (type first-tangent-line) first-tangent-mode)
  (cond ((or (equal first-tangent-mode "plus-plus")
             (equal first-tangent-mode "plus-minus"))
         (cond ((null (plus-tangents originating-vertex))
                (setf (plus-tangents originating-vertex)
                  (list first-tangent-line)))
               (t (setf (plus-tangents originating-vertex)
                    (cons first-tangent-line
                          (plus-tangents originating-vertex)))))
         (cond ((equal first-tangent-mode "plus-plus")
                (cond ((null (minus-tangents terminating-vertex))
                       (setf (minus-tangents terminating-vertex)
                         (list second-tangent-line)))
                      (t (setf (minus-tangents terminating-vertex)
                           (cons second-tangent-line
                                 (minus-tangents terminating-vertex))))))
               (t
                (cond ((null (plus-tangents terminating-vertex))
                       (setf (plus-tangents terminating-vertex)
                         (list second-tangent-line)))
                      (t (setf (plus-tangents terminating-vertex)
                           (cons second-tangent-line
                                 (plus-tangents terminating-vertex)))))))))
        ((or (equal first-tangent-mode "minus-minus")
          (equal first-tangent-mode "minus-plus"))
        (cond ((null (minus-tangents originating-vertex))
                (setf (minus-tangents originating-vertex)
                  (list first-tangent-line)))
               (t (setf (minus-tangents originating-vertex)
                    (cons first-tangent-line
                          (minus-tangents originating-vertex)))))
        (cond ((equal first-tangent-mode "minus-minus")
                (cond ((null (plus-tangents terminating-vertex))
                        (setf (plus-tangents terminating-vertex)
                          (list second-tangent-line)))
                       (t (setf (plus-tangents terminating-vertex)
                            (cons second-tangent-line
                                  (plus-tangents terminating-vertex))))))
               (t
                (cond ((null (minus-tangents terminating-vertex))
                        (setf (minus-tangents terminating-vertex)
                          (list second-tangent-line)))
                       (t (setf (minus-tangents terminating-vertex)
                            (cons second-tangent-line
                                  (minus-tangents terminating-vertex))))))))))

;-----------------------------------------------------------
;   LOCATE-TANGENTS
;-----------------------------------------------------------
(defmethod locate-the-tangents (polygon-list)
 (locate-some-tangents (first polygon-list)
                       (rest polygon-list) polygon-list))

;-----------------------------------------------------------
;   LOCATE-SOME-TANGENTS
;-----------------------------------------------------------
(defmethod locate-some-tangents ((fixed-polygon polygon)
                                  list-of-polygons
                                  obstacle-list)
 (cond ((null list-of-polygons) nil)
     (t (dolist (fixed-vertex (vertice-list fixed-polygon))
          (find-tangents fixed-vertex
                         list-of-polygons
                         obstacle-list))
        (locate-some-tangents (first list-of-polygons)
                              (rest  list-of-polygons)
                              obstacle-list))))

;-----------------------------------------------------------
;   FIND-TANGENTS
;-----------------------------------------------------------
```

```
(defmethod find-tangents (fixed-vertex
                          list-of-polygons
                          obstacle-list)
 (cond ((null list-of-polygons) nil)
      (t (dolist (polygon list-of-polygons)
           (construct-tangents-1 fixed-vertex
                        (cond ((equal (type polygon) "concave")
                               (exterior-vertice-list polygon))
                              (t (vertice-list polygon)))
                        obstacle-list)))))

;------------------------------------------------------
;   CONSTRUCT-TANGENTS
;------------------------------------------------------
(defmethod construct-tangents-1 (fixed-vertex
                                 list-of-exterior-vertices
                                 obstacle-list)
 (locate-all-tangents fixed-vertex
                      list-of-exterior-vertices
                      obstacle-list))

;------------------------------------------------------
;   LOCATE-ALL-TANGENTS
;------------------------------------------------------
(defun locate-all-tangents (fixed-vertex
                            list-of-exterior-vertices
                            obstacle-list)
 (cond ((null list-of-exterior-vertices) nil)
      (t (dolist (current-vertex list-of-exterior-vertices)
           (let ((adjacency-test-result (adjacency-test
                                         fixed-vertex
                                         current-vertex)))
             (cond ((equal adjacency-test-result "adjacent") nil)
                   (t (let* ((from-fixed-vertex-test (check-line current-vertex fixed-vertex))
                             (from-current-vertex-test (check-line fixed-vertex current-vertex))
                             (temp-tangent (make-tangent-line
                                            fixed-vertex
                                            current-vertex)))
                       (cond ((and (equal from-fixed-vertex-test "--")
                                   (equal from-current-vertex-test "++"))
                              (test-verify-and-attach  temp-tangent
                                                       fixed-vertex
                                                       current-vertex
                                                       "plus-plus"
                                                       obstacle-list))
                             ((and (equal from-fixed-vertex-test  "--")
                                   (equal from-current-vertex-test "--"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "minus-plus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "++")
      (equal from-current-vertex-test "--"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "minus-minus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "++")
      (equal from-current-vertex-test "++"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "plus-minus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "0+")
      (equal from-current-vertex-test "-0"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "minus-minus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "-0")
      (equal from-current-vertex-test "-0"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "minus-plus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "-0")
      (equal from-current-vertex-test "0+"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "plus-plus"
                          obstacle-list))
((and (equal from-fixed-vertex-test  "0+")
      (equal from-current-vertex-test "0+"))
 (test-verify-and-attach  temp-tangent
                          fixed-vertex
                          current-vertex
                          "plus-minus"
                          obstacle-list))
```

```
                    (t nil))))))))))

;-----------------------------------------------------------
;   DEFUN CHECK-VISIBILITY-OF-START-TO-GOAL-PATH
;-----------------------------------------------------------
(defun check-visibility-of-start-to-goal-path ()
 (let* ((start-to-goal-line (make-tangent-line *start-point*
                                               *goal-point*))
        (test-result (check-visibility start-to-goal-line
                                        *my-list*)))
  (cond ((equal test-result nil) "no intersection")
        (t "intersection"))))

;-----------------------------------------------------------
;   DEFMETHOD CHECK-VISIBILITY-OF-GOAL
;-----------------------------------------------------------
(defmethod check-visibility-of-goal ((vertex vertex))
 (let* ((vertex-to-goal-line (make-tangent-line vertex
                                                *goal-point*))
        (test-result-1 (check-line-from-vertex vertex
                                                *my-list*))
        (test-result-2 (check-visibility vertex-to-goal-line
                                          *my-list*)))
  (cond ((and (equal test-result-1 "no intersection")
              (equal test-result-2 "no intersection"))
         "no intersection")
        (t "intersection"))))

;-----------------------------------------------------------
;   DEFUN BUILD-TANGENTS
;-----------------------------------------------------------
(defun build-tangents (start polygon-list)
 (find-tangents-from-point *start-point*
                           polygon-list
                           polygon-list))


;;;          SPECIAL FUNCTIONS FOR PROCESSING
;;;          POINT-TO-POLYGON TANGENTS
;-----------------------------------------------------------
;   DEFMETHOD FIND-TANGENTS-FROM-POINT
;-----------------------------------------------------------
(defmethod find-tangents-from-point (fixed-vertex
                                     list-of-polygons
                                     obstacle-list)
 (cond ((null list-of-polygons) nil)
    (t (dolist (polygon list-of-polygons)
         (construct-tangents-from-point fixed-vertex
                                       (cond ((equal (type polygon) "concave")
                                              (exterior-vertice-list polygon))
                                             (t (vertice-list polygon)))
                                              obstacle-list)))))

;-----------------------------------------------------------
;    DEFMETHOD CONSTRUCT-TANGENTS-FROM-POINT
;-----------------------------------------------------------
(defmethod construct-tangents-from-point ((fixed-vertex vertex)
                                          list-of-exterior-vertices
                                          obstacle-list)
 (locate-all-tangents-from-point fixed-vertex
                                 list-of-exterior-vertices
                                 obstacle-list))

;-----------------------------------------------------------
;    DEFMETHOD LOCATE-ALL-TANGENTS-FROM-POINT
;-----------------------------------------------------------
(defmethod locate-all-tangents-from-point ((fixed-vertex  vertex)
                                           list-of-exterior-vertices
                                           obstacle-list)
 (cond ((null list-of-exterior-vertices) nil)
     (t (dolist (current-vertex list-of-exterior-vertices)
          (let ((adjacency-test-result (adjacency-test
                                        fixed-vertex
                                        current-vertex)))
            (let* ((from-fixed-vertex-test (check-line-from-point
                                            current-vertex
                                            fixed-vertex))
                   (from-current-vertex-test (check-line-from-point
                                              fixed-vertex
                                              current-vertex))
                   (temp-tangent (make-tangent-line
                                  fixed-vertex
                                  current-vertex)))
             (cond ((and (equal from-fixed-vertex-test "--")
                         (or (equal from-current-vertex-test "++")
                             (equal from-current-vertex-test "00")))
                    (test-and-attach  temp-tangent
                                      0
                                      0
                                      fixed-vertex
                                      current-vertex
                                      "plus-plus"
                                      obstacle-list))
                   ((and (equal from-fixed-vertex-test  "--")
          (or (equal from-current-vertex-test "--")
             (equal from-current-vertex-test "00")))
  (test-and-attach  temp-tangent
                    0
                    0
                    fixed-vertex
                    current-vertex
                    "plus-plus"
                    obstacle-list))
 ((and (equal from-fixed-vertex-test  "++")
          (or (equal from-current-vertex-test "--")
             (equal from-current-vertex-test "00")))
  (test-and-attach  temp-tangent
                    0
                    0
                    fixed-vertex
                    current-vertex
                    "minus-minus"
                    obstacle-list))
 ((and (equal from-fixed-vertex-test  "++")
          (or (equal from-current-vertex-test "++")
             (equal from-current-vertex-test "00" )))
  (test-and-attach  temp-tangent
                    0
                    0
                    fixed-vertex
                    current-vertex
                    "plus-minus"
                    obstacle-list))
 ((and (equal from-fixed-vertex-test  "0+")
          (or (equal from-current-vertex-test "-0")
             (equal from-current-vertex-test "00")))
  (test-and-attach  temp-tangent
                    0
                    0
                    fixed-vertex
                    current-vertex
                    "minus-minus"
                    obstacle-list))
 ((and (equal from-fixed-vertex-test  "-0")
          (or (equal from-current-vertex-test "-0")
             (equal from-current-vertex-test "00")))
  (test-and-attach  temp-tangent
                    0
                    0
                    fixed-vertex
                    current-vertex
```

```
                                        "plus-plus"
                                        obstacle-list))
                    ((and (equal from-fixed-vertex-test  "-0")
                          (or (equal from-current-vertex-test "0+")
                             (equal from-current-vertex-test "00")))
                     (test-and-attach  temp-tangent
                                        0
                                        0
                                        fixed-vertex
                                        current-vertex
                                        "plus-plus"
                                        obstacle-list))
                    ((and (equal from-fixed-vertex-test  "0+")
                          (or (equal from-current-vertex-test "0+")
                             (equal from-current-vertex-test "00")))
                     (test-and-attach  temp-tangent
                                        0
                                        0
                                        fixed-vertex
                                        current-vertex
                                        "plus-minus"
                                        obstacle-list))
                    (t nil))))))))

;-----------------------------------------------------------
;    DEFMETHOD CHECK-LINE-FROM-POINT
;-----------------------------------------------------------
(defmethod check-line-from-point ((polygon-1-vertex vertex)
                                  (polygon-2-vertex vertex))
 (let* ((predecessor-test (point-position (coordinates (predecessor polygon-1-vertex))
                                  (coordinates polygon-1-vertex)
                                  (coordinates polygon-2-vertex)))
     (successor-test  (point-position (coordinates (successor polygon-1-vertex))
                                  (coordinates polygon-1-vertex)
                                  (coordinates polygon-2-vertex))))
  (cond ((and (< predecessor-test 0)
              (> successor-test  0)) "-+")
        ((and (> predecessor-test 0)
              (< successor-test  0)) "+-")
        ((and (> predecessor-test 0)
              (> successor-test  0)) "++")
        ((and (< predecessor-test 0)
              (< successor-test  0)) "--")
        ((and (= predecessor-test 0)
              (> successor-test  0)) "0+")
     ((and (= predecessor-test 0)
              (< successor-test  0)) "0-")
```

```
    ((and (> predecessor-test 0)
              (= successor-test  0)) "+0")
    ((and (< predecessor-test 0)
              (= successor-test  0)) "-0")
      ((and (= predecessor-test 0)
              (= successor-test  0)) "00"))))


;-------------------------------------------------------
;    DEFMETHOD CHECK-LINE-FROM-VERTEX
;-------------------------------------------------------
(defmethod check-line-from-vertex ((fixed-vertex  vertex)
                                    obstacle-list)
 (let* ((from-current-vertex-test
           (check-line-from-point *goal-point*
                                  fixed-vertex))
 (setf (predecessor *start-point*) *start-point*)
 (setf (successor   *start-point*) *start-point*
        (from-fixed-vertex-test (check-line-from-point
                                   fixed-vertex
                                   *goal-point*)))
   (cond ((and (equal from-fixed-vertex-test "--")
               (or (equal from-current-vertex-test "++")
                 (equal from-current-vertex-test "00")))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "--")
               (or (equal from-current-vertex-test "--")
                 (equal from-current-vertex-test "00")))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "++")
               (or (equal from-current-vertex-test "--")
                 (equal from-current-vertex-test "00")))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "++")
               (or (equal from-current-vertex-test "++")
                 (equal from-current-vertex-test "00" )))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "0+")
               (or (equal from-current-vertex-test "-0")
                 (equal from-current-vertex-test "00")))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "-0")
               (or (equal from-current-vertex-test "-0")
                 (equal from-current-vertex-test "00")))
          "no intersection")
         ((and (equal from-fixed-vertex-test  "-0")
               (or (equal from-current-vertex-test "0+")
           (equal from-current-vertex-test "00")))
 "no intersection")
((and (equal from-fixed-vertex-test  "-0")
        (or (equal from-current-vertex-test "0+")
           (equal from-current-vertex-test "00")))
 "no intersection")
((and (equal from-fixed-vertex-test  "0+")
        (or (equal from-current-vertex-test "0+")
           (equal from-current-vertex-test "00")))
 "no intersection")
(t "intersection"))))
```

```
;;;-----------------------------------------------
;;;
;;;     FILENAME:     CONCAVE.LISP
;;;     AUTHOR:       JERRY A. CRANE
;;;     DATE:         14 AUG 1991
;;;     DESCRIPTION:
;;;  CONTAINS THE FUNCTIONS TO PARTITION
;;;  CONCAVE POLYGONS INTO CONVEX-SUB-
;;;   POLYGONS.
;;;
;;;-----------------------------------------------


(defvar *numbers* (list "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
                        "10" "11" "12" "13" "14" "15"
                        "16" "17" "18" "19" "20" "21"
                        "22" "23" "24" "25" "26" "27"
                        "28" "29" "30" "31" "32" "33"
                        "34" "35" "36" "37" "38" "39"
                        "40" "41" "42" "43" "44" "45"
                        "46" "47" "48" "49" "50"))

;-----------------------------------------------
;   LIST-ADJACENT-VERTICES-TOGETHER
;-----------------------------------------------

(defun list-adjacent-vertices-together (vertice-list)
 (let* ((first-vertex (first vertice-list))
        (last-vertex  (first (last vertice-list)))
        (temp-list   (remove last-vertex vertice-list))
        (test-result  (adjacency-test first-vertex
                                      last-vertex)))
  (cond ((equal test-result "adjacent")
         (setf temp-list (cons last-vertex temp-list))
         (list-adjacent-vertices-together temp-list))
        (t vertice-list))))
;-----------------------------------------------
;   CREATE-CONCAVE-SUB-POLYGONS
;-----------------------------------------------
(defun create-convex-sub-polygons (polygon-list)
 (dolist (poly polygon-list)
  (cond ((equal (type poly) "concave")
         (sub-divide-concave-polygons poly))
        (t))))
```

```
;-------------------------------------------------
;    SUB-DIVIDE-CONCAVE-POLYGONS
;-------------------------------------------------
(defmethod sub-divide-concave-polygons ((poly polygon))
 (setf (exterior-vertice-list poly) (list-adjacent-vertices-together (exterior-vertice-list poly)))
 (let* ((first-vertex (first (exterior-vertice-list poly)))
         (name-list *numbers*)
         (temp-concave-sub-polygon-list (list (make-instance 'convex-sub-polygon
                                                    :name (assign-name (name poly) name-
list)
                                                    :vertice-list (list first-vertex)))))
   (setf (sub-polygon first-vertex) (name (first temp-concave-sub-polygon-list)))
   (setf (parent first-vertex) (first temp-concave-sub-polygon-list))
   (setf (vertex-number first-vertex) 1)
   (setf name-list (rest name-list))
   (dolist (vertex (rest (exterior-vertice-list poly)) temp-concave-sub-polygon-list)
     (let ((result (insert-vertex vertex (vertice-list (first temp-concave-sub-polygon-list)))))
          (cond ((not (null result))
                 (setf (vertice-list (first temp-concave-sub-polygon-list)) result)
                 (setf (parent (first (vertice-list (first temp-concave-sub-polygon-list))))
                              (first temp-concave-sub-polygon-list)))
                (t (setf temp-concave-sub-polygon-list
                      (cons (make-instance 'convex-sub-polygon
                               :name (assign-name (name poly) name-list)
                               :vertice-list (list vertex))
                             temp-concave-sub-polygon-list))
                    (setf name-list (rest name-list))
                    (setf (vertex-number (first (vertice-list (first temp-concave-sub-polygon-list)))) 1)
                    (setf (parent (first (vertice-list (first temp-concave-sub-polygon-list))))
                              (first temp-concave-sub-polygon-list))
                    (setf (sub-polygon (first (vertice-list (first temp-concave-sub-polygon-
list))))
                      (name (first temp-concave-sub-polygon-list)))))))
   (setf (sub-polygons poly) temp-concave-sub-polygon-list)))
```

```
;------------------------------------------------
;    INSERT-A-VERTEX
;------------------------------------------------
(defmethod insert-vertex ((current-vertex vertex)
                          poly)
 (let* ((test-vertex (first poly))
        (current-number-of-vertices (length poly))
        (adjacency-test-result (adjacency-test current-vertex
                                               test-vertex)))
  (cond ((and (equal current-number-of-vertices 4)
              (equal adjacency-test-result "adjacent")) nil)
      ((and (<= current-number-of-vertices 4)
              (equal adjacency-test-result "adjacent"))
         (setf (sub-polygon current-vertex) (sub-polygon test-vertex))
         (setf (vertex-number current-vertex)
          (+ current-number-of-vertices 1))
         (setf poly (cons current-vertex poly)))
      (t nil))))

;------------------------------------------------
;    ASSIGN-NAME
;------------------------------------------------
(defun assign-name (name name-list)
 (concatenate 'string name (first name-list)))
(setf *new-list* *my-list*)

;------------------------------------------------
;    COLLECT-TANGENTS
;------------------------------------------------
(defun collect-tangents (polygon-list)
 (dolist (poly polygon-list)
  (let ((plus-tangent-list nil)
        (minus-tangent-list nil))
   (dolist (vertex (vertice-list poly) plus-tangent-list)
       (cond ((null (first (plus-tangents vertex))) nil)
          (t
           (dolist (tangent (plus-tangents vertex))
               (cond ((null plus-tangent-list)
                      (setf plus-tangent-list (list tangent)))
                     (t (setf plus-tangent-list (cons tangent plus-tangent-list))))))))
   (dolist (vertex (vertice-list poly) minus-tangent-list)
       (cond ((null (first (minus-tangents vertex))) nil)
          (t
           (dolist (tangent (minus-tangents vertex))
               (cond ((null minus-tangent-list)
                      (setf minus-tangent-list (list tangent)))
```

```
                   (t (setf minus-tangent-list (cons tangent minus-tangent-list))))))))
   (setf (plus-tangents poly) plus-tangent-list)
   (setf (minus-tangents poly) minus-tangent-list)
   (cond ((equal (type poly) "concave")
          (collect-sub-polygon-tangents (sub-polygons poly)))
         (t nil)))))

;---------------------------------------------------
;    COLLECT-SUB-POLYGON-TANGENTS
;---------------------------------------------------
(defun collect-sub-polygon-tangents (polygon-list)
 (dolist (poly polygon-list)
  (let ((plus-tangent-list nil)
        (minus-tangent-list nil))
   (dolist (vertex (vertice-list poly) plus-tangent-list)
      (cond ((null (first (plus-tangents vertex))) nil)
            (t
             (dolist (tangent (plus-tangents vertex))
                (cond ((null plus-tangent-list)
                       (setf plus-tangent-list (list tangent)))
                      (t (setf plus-tangent-list (cons tangent plus-tangent-list))))))))
   (dolist (vertex (vertice-list poly) minus-tangent-list)
      (cond ((null (first (minus-tangents vertex))) nil)
            (t
             (dolist (tangent (minus-tangents vertex))
                (cond ((null minus-tangent-list)
                       (setf minus-tangent-list (list tangent)))
                      (t (setf minus-tangent-list (cons tangent minus-tangent-list))))))))
   (setf (plus-tangents poly) plus-tangent-list)
   (setf (minus-tangents poly) minus-tangent-list))))
```

gemini:crane

stdin

Wed Sep 18 22:25:48 1991

ps / LaserWriter II NTX

```
ps gemini:crane  Job: stdin  Date: Wed Sep 18 22:25:48 1991

ps gemini:crane  Job: stdin  Date: Wed Sep 18 22:25:48 1991

ps gemini:crane  Job: stdin  Date: Wed Sep 18 22:25:48 1991

ps gemini:crane  Job: stdin  Date: Wed Sep 18 22:25:48 1991
```

```
;;;-----------------------------------------------
;;;
;;;       FILENAME:     NEW-PATH.LISP
;;;       AUTHOR:       JERRY A. CRANE
;;;       DATE:         14 AUG 1991
;;;
;;;       DESCRIPTION:
;;; Contains the functions and methods for
;;; path searching.  Uses a modified Dijkstra
;;; Algorithm to search for the shortest path.
;;;
;;;-----------------------------------------------

;--------------------------------------------------------
;    DEFUN RESET-WORLD
;--------------------------------------------------------
(defun reset-world ()
 (zero-modes)
 (setf *initial-path-list* nil)
 (setf *polygon-mode-list* nil)
 (setf *goal-found-flag* "false")
 (setf *my-bitmap-stream-3* *bitmap-copy*)
 (cw:bitblt *my-bitmap-stream-3* 0 0 *win-3* 0 0)
 (get-start-and-goal-coordinates-2)
 (build-tangents *start-point* *my-list*)
 (shortest-paths))

(defvar *start* nil)
(defvar *goal*  nil)

(defvar *initial-path-list* nil)
(defvar *polygon-mode-list* nil)
(defvar *goal-found-flag* "false")
(defvar *new* nil)
(defvar *start-point*)
(defvar *goal-point*)
(defvar *error-flag*)
(defvar *stuff* "run")
(setq *bitmap-copy* *my-bitmap-stream-3*)

;--------------------------------------------------------
;    DEFUN GET-START-AND-GOAL-COORDINATES
;--------------------------------------------------------
(defun get-start-and-goal-coordinates-2 ()
 (format t "ENTER THE START POINT COORDINATES: (X Y) ")
 (setf *start* (read))
 (format t "ENTER THE GOAL POINT COORDINATES:  (X Y) ")
```

```
(setf *goal* (read))
(setf *start-point* (make-vertex (first *start*)
                                 (second *start*)
                                 "start"))
(setf *goal-point* (make-vertex (first *goal*)
                                (second *goal*)
                                 "goal"))
(cond  ((equalp *display* "yes")
        (draw-vertex *win-3* *my-bitmap-stream-3* *start-point*)
        (label-vertex *win-3* *my-bitmap-stream-3* "START" *start-point*)))
(cond ((equalp *display* "yes")
       (draw-vertex *win-3* *my-bitmap-stream-3* *goal-point*)
       (label-vertex *win-3* *my-bitmap-stream-3* "GOAL" *goal-point*)))
(setf (predecessor *start-point*) *start-point*)
(setf (successor   *start-point*) *start-point*)
(setf (predecessor *goal-point*) *goal-point*)
(setf (successor   *goal-point*) *goal-point*)
(setf (parent-name *start-point*) "start")
(setf (parent-name *goal-point*) "goal")
(zero-modes))

;-----------------------------------------------------------
;   DEFMETHOD EXTEND-PATH-LIST
;-----------------------------------------------------------
(defmethod extend-path-list ((current-path path-header)
                             polygon-list)
 (let ((mode (path-mode current-path)))
  (cond ((equal mode "plus")
         (let* ((current-path-node (plus-mode (path-end-point
                                               current-path)))
                (new-paths (path-expansion current-path-node
                                           (path-end-point current-path)
                                           mode
                                           polygon-list))
                (new-path-list nil))))
        ((equal mode "minus")
         (let* ((current-path-node (minus-mode (path-end-point
                                                current-path)))
                (new-paths (path-expansion current-path-node
                                           (path-end-point current-path)
                                           mode
                                           polygon-list))
                (new-path-list nil))))))

;-----------------------------------------------------------
;   DEFMETHOD EXTEND-PATH-LIST
;-----------------------------------------------------------
```

```
(defmethod expand-path-list ((current-polygon-path-node path-node)
                              polygon-list)

 (path-expansion current-polygon-path-node
                 (from-polygon current-polygon-path-node)
                 (path-mode current-polygon-path-node)
                 polygon-list))
```

```
;-----------------------------------------------------------
;   DEFMETHOD PATH-EXPANSION
;-----------------------------------------------------------
(defmethod path-expansion ((current-path-node path-node)
                           (path-end-point polygon)
                           current-path-mode
                           polygon-list)

 (let* ((landing-vertex (landing-vertex current-path-node))
        (from-vertex   (from-vertex   current-path-node))
        (tangent-angle  (beta1 from-vertex landing-vertex))
        (allowable-tangent-paths nil)
        (edge-traversal-cost 0)
        (path-area (path-area current-path-node))
        (current-path-area path-area))
        ;...............................................................
        ; PROCESS THE PLUS-MODE SLOT OF THE
        ; POLYGON OF THE LANDING-VERTEX
        ; COLLECT PLUS-TANGENTS FROM LANDING
        ; VERTEX LEFT OF INCOMING TANGENT
        ; AS WELL AS THE VERTEX WHICH IS THE
        ; PREDECESSOR OF THE LANDING VERTEX
        ;......................................................................
  (cond ((equal current-path-mode "plus")
         (cond ((equal (check-visibility-of-goal landing-vertex) "no intersection")
                (setf *goal-found-flag* "true")
                (draw-dark-line-3 *win-3*
                                  *my-bitmap-stream-3*
                                  landing-vertex
                                  *goal-point*)))

         (dolist (tangent (plus-tangents landing-vertex) allowable-tangent-paths)
          (let ((normal-angle (- (angle tangent) tangent-angle)))
           (setf current-path-area
                 (+ current-path-area
                    (compute-path-area (end-point-1 tangent)
                                       (end-point-2 tangent))))
           (cond ((and (>= normal-angle 0) (<= normal-angle 180))
                  (cond ((null allowable-tangent-paths)
                         (setf allowable-tangent-paths
                          (list (list tangent
                                      edge-traversal-cost
                                      current-path-area))))
                        (t (setf allowable-tangent-paths
                               (cons (list tangent
                                           edge-traversal-cost
```

```
                                   current-path-area)
                              allowable-tangent-paths))))))))
  (let* ((next-vertex (successor landing-vertex)))
    ; COMPUTE EDGE TRAVERSAL COST
    (setf edge-traversal-cost
      (straight-line-distance landing-vertex
                              next-vertex))
    ; COMPUTE EDGE TRAVERSAL PATH AREA, ADD TO PATH
      ; AREA TO GET TO THIS POINT
    (setf current-path-area (+ path-area
                                 (compute-path-area landing-vertex
                                                    next-vertex)))
    (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
                (equal *goal-found-flag* "false"))
           (setf *goal-found-flag* "true")
           (draw-dark-line-3 *win-3*
                             *my-bitmap-stream-3*
                             next-vertex
                             *goal-point*)))
    (dolist (tangent (plus-tangents next-vertex) allowable-tangent-paths)
      ; COMPUTE PATH AREA FOR POLYGON,
        ; EDGE, AND TANGENT TO NEXT POLYGON
      (let ((new-path-area (+ (compute-path-area (end-point-1 tangent)
                                                 (end-point-2 tangent))
                              current-path-area)))
        (cond ((null allowable-tangent-paths)
               (setf allowable-tangent-paths
                 (list (list tangent edge-traversal-cost new-path-area))))
              (t (setf allowable-tangent-paths
                   (cons (list tangent edge-traversal-cost new-path-area)
                         allowable-tangent-paths))))))))
  ;.........................................................................
; PROCESS THE MINUS-MODE SLOT OF THE
; POLYGON OF THE LANDING-VERTEX
; COLLECT MINUS-TANGENTS FROM LANDING
;  VERTEX RIGHT OF INCOMING TANGENT
; AS WELL AS THE VERTEX WHICH IS THE
; SUCCESSOR OF THE LANDING VERTEX
;.........................................................................
 ((equal current-path-mode "minus")
  (cond ((equal (check-visibility-of-goal landing-vertex) "no intersection")
         (setf *goal-found-flag* "true")
         (draw-dark-line-3 *win-3*
                           *my-bitmap-stream-3*
                           landing-vertex
                           *goal-point*)))
  (dolist (tangent (minus-tangents landing-vertex) allowable-tangent-paths)
            (let ((normal-angle (- (angle tangent) tangent-angle)))
                    (setf current-path-area
                      (+ current-path-area (compute-path-area (end-point-1 tangent)
                                                               (end-point-2 tangent))))
              (cond ((<= normal-angle 0)
                      (cond ((null allowable-tangent-paths)
                               (setf allowable-tangent-paths
                                 (list (list tangent edge-traversal-cost current-path-area))))
                              (t (setf allowable-tangent-paths
                                      (cons (list tangent edge-traversal-cost current-path-area)
                                          allowable-tangent-paths))))))))
          (let* ((next-vertex (predecessor landing-vertex)))
            ; COMPUTE EDGE TRAVERSAL COST
            (setf edge-traversal-cost (straight-line-distance landing-vertex next-vertex))
            ; COMPUTE EDGE TRAVERSAL PATH AREA,
              ;  ADD TO PATH AREA TO GET TO THIS POINT
            (setf current-path-area
              (+ path-area (compute-path-area landing-vertex
                                              next-vertex)))
            (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
                          (equal *goal-found-flag* "false"))
                    (setf *goal-found-flag* "true")
                    (draw-dark-line-3 *win-3*
                                      *my-bitmap-stream-3*
                                      next-vertex
                                      *goal-point*)))
            (dolist (tangent (minus-tangents next-vertex) allowable-tangent-paths)
              ; COMPUTE PATH AREA FOR POLYGON,
                ;  EDGE, AND TANGENT TO NEXT POLYGON
              (let ((new-path-area (+ (compute-path-area (end-point-1 tangent)
                                                          (end-point-2 tangent))
                                      current-path-area)))
              (cond ((null allowable-tangent-paths)
                      (setf allowable-tangent-paths
                        (list (list tangent edge-traversal-cost new-path-area))))
                     (t (setf allowable-tangent-paths
                          (cons (list tangent edge-traversal-cost new-path-area)
                                allowable-tangent-paths)))))))))
  (setf *new* allowable-tangent-paths)

  (cond ((equalp *display* "yes")
          (dolist (tangent *new*)
            (build-subsequent-path-node (first tangent)
                                        (second tangent)
                                        (third tangent)
                                        (end-point-1 (first tangent))
```

```
                                               (end-point-2 (first tangent))
                                               (type (first tangent)))))
                 (t
                  (dolist (tangent *new*)
                   (build-subsequent-path-node (first tangent)
                                               (second tangent)
                                               (third tangent)
                                               (end-point-1 (first tangent))
                                               (end-point-2 (first tangent))
                                               (type (first tangent))))))))


;;;=================================================================;
;;      DEFMETHOD PATH-EXPANSION  (CONVEX-SUB-POLYGONS)
;;;=================================================================
(defmethod path-expansion ((current-path-node path-node)
                           (path-end-point convex-sub-polygon)
                           current-path-mode
                           polygon-list)
 (let* ((landing-vertex (landing-vertex current-path-node))
        (from-vertex   (from-vertex   current-path-node))
        (tangent-angle  (beta1 from-vertex landing-vertex))
        (allowable-tangent-paths nil)
        (edge-traversal-cost 0)
        (path-area (path-area current-path-node))
        (current-path-area path-area))
        ;.................................................................
        ; PROCESS THE PLUS-MODE SLOT OF THE CONVEX-SUB-POLYGON
        ; OF THE LANDING-VERTEX
        ; FIRST STEP IS TO COLLECT THE MINUS TANGENTS RIGHT OF
        ; THE INCOMING TANGENT
        ;.................................................................
  (cond ((equal current-path-mode "plus")
         (cond ((equal (check-visibility-of-goal landing-vertex) "no intersection")
                (setf *goal-found-flag* "true")
                (draw-dark-line-3 *win-3*
                                  *my-bitmap-stream-3*
                                  landing-vertex
                                  *goal-point*)))
         (dolist (tangent (plus-tangents landing-vertex) allowable-tangent-paths)
          (let ((normal-angle (- (angle tangent) tangent-angle)))
           (setf current-path-area
                (+ path-area (compute-path-area (end-point-1 tangent)
                                                (end-point-2 tangent))))
           (cond ((and (>= normal-angle 0) (<= normal-angle 180))
                  (cond ((null allowable-tangent-paths)
                         (setf allowable-tangent-paths
                    (list (list tangent
                                edge-traversal-cost
                                current-path-area))))
                   (t (setf allowable-tangent-paths
                        (cons (list tangent
                                    edge-traversal-cost
                                    current-path-area)
                              allowable-tangent-paths))))))))
       ;.......................................................
       ; BEGIN PROCESSING ALONG ADJACENT VERTICES USING
       ; THE SUCCESSOR SLOT
       ;.......................................................
 (let* ((next-vertex (successor landing-vertex))
        (current-parent (parent landing-vertex))
        (last-vertex-examined  next-vertex))
  ; COMPUTE EDGE TRAVERSAL COST
  (setf edge-traversal-cost (straight-line-distance landing-vertex
                                                    next-vertex))
  ; COMPUTE EDGE TRAVERSAL PATH AREA, ADD TO
  ; PATH AREA TO GET TO THIS POINT
  (setf current-path-area
   (+ path-area (compute-path-area landing-vertex
                                   next-vertex)))
  (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
              (equal (vertex-type next-vertex)
                     (vertex-type landing-vertex)))
         (setf *goal-found-flag* "true")
         (draw-dark-line-3 *win-3* *my-bitmap-stream-3* next-vertex *goal-point*)))
  (loop
   ; BEGIN PROCESSING ADJACENT VERTICES UNTIL
   ; ONE IS FOUND BELONGING TO
   ; A DIFFERENT SUB-POLYGON
   (when (not (equal (parent landing-vertex) (parent next-vertex)))
        (return allowable-tangent-paths))
   (dolist (tangent (plus-tangents next-vertex) allowable-tangent-paths)
        (let ((new-path-area (+ current-path-area
                                (compute-path-area (end-point-1 tangent)
                                                   (end-point-2 tangent)))))
         (cond ((null allowable-tangent-paths)
                (setf allowable-tangent-paths
                 (list (list tangent
                             edge-traversal-cost
                             current-path-area))))
               (t (setf allowable-tangent-paths
                   (cons (list tangent
                               edge-traversal-cost
                               current-path-area)
                        allowable-tangent-paths))))))
            (setf last-vertex-examined next-vertex)
            (setf next-vertex (successor next-vertex))
         ; INCREMENT EDGE TRAVERSAL COST FOR THE NEXT EDGE
            (setf edge-traversal-cost
             (+ edge-traversal-cost
                  (straight-line-distance last-vertex-examined
                                             next-vertex)))
       ; INCREMENT PATH AREA TO INCLUDE THE NEW EDGE
            (setf current-path-area
             (+ current-path-area
                  (compute-path-area last-vertex-examined
         next-vertex))))
      (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
                   (equal *goal-found-flag* "false")
                   (equal (vertex-type next-vertex)
                          (vertex-type landing-vertex)))
             (setf *goal-found-flag* "true")
             (draw-dark-line-3 *win-3*
                               *my-bitmap-stream-3*
                               next-vertex
                               *goal-point*)))
            ;.......................................................
        ;  LAST VERTEX OF THIS SUB-POLYGON FOUND.
        ; CHECK THE NEXT ADJACENT VERTEX
        ;  IN CASE WE NEED TO TRAVERSE ALONG AN EDGE.
        ; THIS IS THE RESULT OF THE
        ;  WAY THAT CONCAVE SUB-POLYGONS HAVE BEEN DIVIDED
        ;.......................................................
      (let ((adjacency-test-result (adjacency-test last-vertex-examined
                                                       next-vertex))
            (temp-tangent nil))
       (cond ((and (equal adjacency-test-result "adjacent")
                    (equal (vertex-type next-vertex)
                           (vertex-type landing-vertex)))
              (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
                          (equal (vertex-type next-vertex)
                                 (vertex-type landing-vertex)))
                     (setf *goal-found-flag* "true")
                     (draw-dark-line-3 *win-3*
                                       *my-bitmap-stream-3*
                                       next-vertex
                                       *goal-point*)))
              (setf temp-tangent
                   (make-tangent-line last-vertex-examined
                                      next-vertex
                                      "plus-plus"))
              (setf (distance temp-tangent)
                   (straight-line-distance last-vertex-examined
                                                  next-vertex))
              (cond ((null allowable-tangent-paths)
                       (setf allowable-tangent-paths
                      (list (list temp-tangent
                                     edge-traversal-cost
                                     current-path-area))))
                      (t (setf allowable-tangent-paths
                            (cons (list temp-tangent
                                      edge-traversal-cost
                                      current-path-area)
                                allowable-tangent-paths)))))))))

   ;..............................................................
     ; PROCESS THE MINUS-MODE SLOT OF
     ; THE CONVEX-SUB-POLYGON OF THE LANDING-VERTEX
   ; FIRST STEP IS TO COLLECT THE MINUS TANGENTS
     ; RIGHT OF THE INCOMING TANGENT
     ;.........................................................................
  ((equal current-path-mode "minus")
   (cond ((equal (check-visibility-of-goal landing-vertex) "no intersection")
            (setf *goal-found-flag* "true")
            (draw-dark-line-3 *win-3*
                                *my-bitmap-stream-3*
                                landing-vertex
                                *goal-point*)))
   (dolist (tangent (minus-tangents landing-vertex) allowable-tangent-paths)
     (let ((normal-angle (- (angle tangent) tangent-angle)))
       (setf current-path-area (+ path-area (compute-path-area (end-point-1 tangent)
                                                               (end-point-2 tangent))))
       (cond ((and (<= normal-angle 0) (>= normal-angle -180))
              (cond ((null allowable-tangent-paths)
                       (setf allowable-tangent-paths
                         (list (list tangent edge-traversal-cost current-path-area))))
                      (t (setf allowable-tangent-paths
                            (cons (list tangent edge-traversal-cost current-path-area)
                               allowable-tangent-paths))))))))
      ;.........................................................................
      ; BEGIN PROCESSING ALONG ADJACENT
        ; VERTICES USING THE PREDECESSOR SLOT
      ;.........................................................................
   (let* ((next-vertex (predecessor landing-vertex))
          (current-parent (parent landing-vertex))
          (last-vertex-examined   next-vertex))
     ; COMPUTE EDGE TRAVERSAL COST
     (setf edge-traversal-cost (straight-line-distance landing-vertex
                                             next-vertex))
; COMPUTE EDGE TRAVERSAL PATH AREA,
  ;  ADD TO PATH AREA TO GET TO THIS POINT
(setf current-path-area
 (+ path-area (compute-path-area landing-vertex
                                  next-vertex)))
(cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
            (equal (vertex-type next-vertex)
                   (vertex-type landing-vertex)))
       (setf *goal-found-flag* "true")
       (draw-dark-line-3 *win-3*
                         *my-bitmap-stream-3*
                         next-vertex
                         *goal-point*)))

(loop
 ; BEGIN PROCESSING ADJACENT VERTICES
   ;  UNTIL ONE IS FOUND BELONGING TO
 ; A DIFFERENT SUB-POLYGON
 (when (not (equal (parent landing-vertex) (parent next-vertex)))
       (return allowable-tangent-paths))
 (dolist (tangent (minus-tangents next-vertex) allowable-tangent-paths)
     (let ((new-path-area
            (+ current-path-area (compute-path-area (end-point-1 tangent)
                                                    (end-point-2 tangent)))))
      (cond ((null allowable-tangent-paths)
             (setf allowable-tangent-paths
              (list (list tangent edge-traversal-cost current-path-area))))
            (t (setf allowable-tangent-paths
                (cons (list tangent edge-traversal-cost current-path-area)
                      allowable-tangent-paths))))))
 (setf last-vertex-examined next-vertex)
 (setf next-vertex (predecessor next-vertex))
 ; INCREMENT EDGE TRAVERSAL COST FOR THE NEXT EDGE
 (setf edge-traversal-cost
     (+ edge-traversal-cost
       (straight-line-distance last-vertex-examined
                               next-vertex)))
 ; INCREMENT PATH AREA TO INCLUDE THE NEW EDGE
 (setf current-path-area
     (+ current-path-area (compute-path-area last-vertex-examined
                                             next-vertex)))
 (cond ((and (equal (check-visibility-of-goal next-vertex) "no intersection")
             (equal (vertex-type next-vertex)
                    (vertex-type landing-vertex)))
       (setf *goal-found-flag* "true")
       (draw-dark-line-3 *win-3*
                                             *my-bitmap-stream-3*
                                             next-vertex
                                             *goal-point*))))
                            ;..........................................................................
                            ; LAST VERTEX OF THIS SUB-POLYGON
                              ; FOUND. CHECK THE NEXT ADJACENT VERTEX
                            ; IN CASE WE NEED TO TRAVERSE ALONG AN EDGE.
                              ; THIS IS THE RESULT OF THE
                            ; WAY THAT CONCAVE SUB-POLYGONS HAVE BEEN DIVIDED
                            ;..........................................................................
                    (let ((adjacency-test-result (adjacency-test last-vertex-examined
                                                                  next-vertex))
                          (temp-tangent nil))

                     (cond ((and (equal adjacency-test-result "adjacent")
                                 (equal (vertex-type next-vertex)
                                        (vertex-type landing-vertex)))
                            (setf temp-tangent (make-tangent-line last-vertex-examined
                                                                   next-vertex
                                                                   "minus-minus"))
                            (setf (distance temp-tangent)
                               (straight-line-distance last-vertex-examined
                                                        next-vertex))
                            (cond ((null allowable-tangent-paths)
                                   (setf allowable-tangent-paths
                                    (list (list temp-tangent edge-traversal-cost current-path-area))))
                                  (t (setf allowable-tangent-paths
                                       (cons (list temp-tangent edge-traversal-cost current-path-area)
                                          allowable-tangent-paths))))))))))
               (setf *new* allowable-tangent-paths)
               (cond ((equalp *display* "yes")
                      (dolist (tangent *new*)
                       (build-subsequent-path-node (first tangent)
                                                   (second tangent)
                                                   (third tangent)
                                                   (end-point-1 (first tangent))
                                                   (end-point-2 (first tangent))
                                                   (type (first tangent)))))
                     (t
                      (dolist (tangent *new*)
                       (build-subsequent-path-node (first tangent)
                                                   (second tangent)
                                                   (third tangent)
                                                   (end-point-1 (first tangent))
                                                   (end-point-2 (first tangent))
                                                   (type (first tangent))))))))
```

```
;-------------------------------------------------------
;   DEFMETHOD TEST-AND-ATTACH
;-------------------------------------------------------
(defmethod test-and-attach ((temp-tangent tangent-line)
                            edge-traversal-cost
                            path-area
                            (fixed-vertex vertex)
                            (current-vertex vertex)
                            tangent-mode
                            obstacle-list)

 (let ((intersection-test (verify-tangent-line-visibility
                            temp-tangent
                            obstacle-list)))
  (cond ((equal intersection-test "valid-tangent-line")

         (build-initial-path-node temp-tangent
                                  edge-traversal-cost
                                  path-area
                                  fixed-vertex
                                  current-vertex
                                  tangent-mode)))))

;-------------------------------------------------------
;   DEFUN ZERO-MODES
;-------------------------------------------------------
(defun zero-modes ()
 (dolist (polygon *my-list*)
  (setf (minus-mode polygon) nil)
  (setf (plus-mode polygon) nil))
 (dolist (polygon *my-list*)
  (cond ((equal (type polygon) "concave")
         (dolist (thing (sub-polygons polygon))
          (setf (minus-mode thing) nil)
          (setf (plus-mode thing) nil))))))

;-------------------------------------------------------
;   DEFMETHOD BUILD-INITIAL-PATH-NODE
;-------------------------------------------------------

(defmethod build-initial-path-node ((first-tangent-line tangent-line)
                                    edge-traversal-cost
                                    path-area
                                    (originating-vertex vertex)
                                    (terminating-vertex vertex)
                                    first-tangent-mode)
```

```
(if (equalp *display* "yes")
   (draw-dark-line-6 *win-3* *my-bitmap-stream-3* first-tangent-line))

(let* ((ending-polygon (parent terminating-vertex))
         (current-path (make-path-header ending-polygon)))
  (cond ((and (or (equal first-tangent-mode "plus-plus")
                        (equal first-tangent-mode "minus-plus"))
                    (null (plus-mode ending-polygon))
                    (equal *stuff* "run"))
            (setf (plus-mode ending-polygon)
              (make-path-node terminating-vertex
                                    originating-vertex
                                    ending-polygon
                                    first-tangent-mode
                                    (+ (straight-line-distance originating-vertex
                                                                          terminating-vertex)
                                        edge-traversal-cost)))
            ;----------- SETTING VALUES IN THE PATH HEADER ------------------
            (setf (path-end-point current-path) ending-polygon)
            (setf (path-cost current-path) (cost (plus-mode ending-polygon)))
            (setf (distance current-path)
              (straight-line-distance terminating-vertex
                                               *goal-point*))
            (setf (total-path-cost current-path)
              (+ (path-cost current-path)
                  (distance current-path)))
            (setf (symbolic-path current-path)
              (list "START" (name (path-end-point current-path)) "+"))
            (setf (path-mode current-path) "plus")
            ;----------- SETTING VALUES IN THE PATH NODE --------------------
            ; PATH-MODE
            (setf (path-mode (plus-mode ending-polygon)) "plus")
            ; SYMBOLIC PATH
            (setf (path (plus-mode ending-polygon))
              (list "START" (name ending-polygon) "+"))
            ; DISTANCE SLOT
            (setf (distance (plus-mode ending-polygon))
              (straight-line-distance terminating-vertex
                                               *goal-point*))
            ; TOTAL-PATH-COST SLOT
            (setf (total-path-cost (plus-mode ending-polygon))
              (+ (cost (plus-mode ending-polygon))
                  (distance (plus-mode ending-polygon))))
            ; PATH-AREA SLOT
            (setf (path-area (plus-mode ending-polygon)) path-area)
            ;--------------------------------------------------------------
            (cond ((null *initial-path-list*)
          (setf *initial-path-list* (list current-path)))
         (t (setf *initial-path-list*
             (cons current-path *initial-path-list*))))
 (cond ((null *polygon-mode-list*)
          (setf *polygon-mode-list* (list (plus-mode ending-polygon))))
         (t (setf *polygon-mode-list*
             (cons (plus-mode ending-polygon) *polygon-mode-list*)))))
((and (or (equal first-tangent-mode "minus-minus")
           (equal first-tangent-mode "plus-minus"))
        (null (minus-mode ending-polygon))
        (equal *stuff* "run"))
 (setf (minus-mode ending-polygon)
  (make-path-node terminating-vertex
                   originating-vertex
                   ending-polygon
                   first-tangent-mode
                   (+ (straight-line-distance originating-vertex
                                               terminating-vertex)
                      edge-traversal-cost)))
;---------- SETTING VALUES IN THE PATH HEADER ---------------
(setf (path-end-point current-path) ending-polygon)
(setf (path-cost current-path) (cost (minus-mode ending-polygon)))
(setf (distance current-path) (straight-line-distance terminating-vertex
                                                       *goal-point*))
(setf (total-path-cost current-path) (+ (path-cost current-path)
                                       (distance current-path)))
(setf (symbolic-path current-path)
 (list "START" (name (path-end-point current-path)) "-"))
(setf (path-mode current-path) "minus")

;---------- SETTING VALUES IN THE PATH NODE -------------------
; PATH-MODE
(setf (path-mode (minus-mode ending-polygon)) "minus")
; SYMBOLIC PATH
(setf (path (minus-mode ending-polygon))
 (list "START" (name ending-polygon) "-"))
; DISTANCE SLOT
(setf (distance (minus-mode ending-polygon))
 (straight-line-distance terminating-vertex *goal-point*))
; TOTAL-PATH-COST SLOT
(setf (total-path-cost (minus-mode ending-polygon))
 (+ (cost (minus-mode ending-polygon))
     (distance (minus-mode ending-polygon))))
; PATH-AREA SLOT
(setf (path-area (minus-mode ending-polygon)) path-area)
;-----------------------------------------------------------------------------------
(cond ((null *initial-path-list*)
                    (setf *initial-path-list* (list current-path)))
                   (t (setf *initial-path-list*
                       (cons current-path *initial-path-list*))))

            (cond ((null *polygon-mode-list*)
                    (setf *polygon-mode-list* (list (minus-mode ending-polygon))))
                   (t (setf *polygon-mode-list*
                       (cons (minus-mode ending-polygon) *polygon-mode-list*))))))))

;-------------------------------------------------------
;    DEFMETHOD BUILD-SUBSEQUENT-PATH-NODE
;-------------------------------------------------------

(defmethod build-subsequent-path-node ((first-tangent-line tangent-line)
                                          edge-traversal-cost
                                          path-area
                                          (originating-vertex vertex)
                                          (terminating-vertex vertex)
                              tangent-mode)

 (let* ((ending-polygon (parent terminating-vertex))
         (current-path (make-path-header ending-polygon))
         (previous-polygon-cost 0)
         (tangent-length (distance first-tangent-line))
      (previous-polygon-path-node nil))
  ;.................................................................
  ; DETERMINE WHETHER TO ACCESS PLUS-MODE OR
  ; MINUS MODE COST AND PATH AREA COST. SET
  ; THESE COSTS EQUAL TO PREVIOUS-POLYGON-COST
  ; AND PREVIOUS-POLYGON-PATH-AREA.
  ;.................................................................
  (cond ((or (equal tangent-mode "plus-plus")
               (equal tangent-mode "plus-minus"))
            (setf previous-polygon-cost
              (cost (plus-mode (parent originating-vertex))))
            (setf previous-polygon-path-area
              (path-area (plus-mode (parent originating-vertex))))
        (setf previous-polygon-path-node
              (plus-mode (parent originating-vertex))))
           (t
            (setf previous-polygon-cost
              (cost (minus-mode (parent originating-vertex))))
            (setf previous-polygon-path-area
              (path-area (minus-mode (parent originating-vertex))))
        (setf previous-polygon-path-node (minus-mode (parent originating-vertex)))))

        ;.............................................................
      ; LANDING MODE IS PLUS SO MUST DETERMINE WHETHER TO BUILD A
         ; PLUS-MODE PATH-NODE BASED ON WHETHER VISITED ALREADY OR NOT.
      ;..................................................................
(cond ((or (equal tangent-mode "plus-plus")
           (equal tangent-mode "minus-plus"))
          ; CURRENT POLYGON PLUS MODE NOT YET VISITED
      (cond ((null (plus-mode ending-polygon))
             (build-plus-path-node originating-vertex
                                   terminating-vertex
                                   previous-polygon-path-node
                                   previous-polygon-cost
                                   previous-polygon-path-area
                                   edge-traversal-cost
                                   tangent-mode
                                   tangent-length
                                   ending-polygon
                                   path-area))
             ; CURRENT POLYGON PLUS MODE PREVIOUSLY VISITED USING
             ; THE SAME LANDING/TERMINATING VERTEX.  SIMPLY COMPARE
             ; PATH COST.  PLUS PATH NODE GETS THE VALUE OF THE
                   ; PATH WITH THE LOWEST PATH COST.
            ((eq terminating-vertex
                 (landing-vertex (plus-mode ending-polygon)))
                   ; NEW PATH COST LESS THAN CURRENT PATH COST
             (cond ((< (+ previous-polygon-cost
                          tangent-length
                          edge-traversal-cost)
                       (cost (plus-mode ending-polygon)))
                    (erase-dark-line-3 *win-3*
                                       *my-bitmap-stream-3*
                                       (from-vertex (plus-mode ending-polygon))
                                       (landing-vertex (plus-mode ending-polygon)))
                    (build-plus-path-node originating-vertex
                                          terminating-vertex
                                          previous-polygon-path-node
                                          previous-polygon-cost
                                          previous-polygon-path-area
                                          edge-traversal-cost
                                          tangent-mode
                                          tangent-length
                                          ending-polygon
                                          path-area))))
             ; CURRENT POLYGON PLUS NODE PREVIOUSLY VISITED USING
             ; DIFFERENT LANDING/TERMINATING VERTICES. MUST CONSIDER
             ; PATH AREA TO DETERMINE WHICH PATH IS UPSTREAM/DOWNSTREAM.
             ; MUST ADD EDGE TRAVERSAL COST TO DOWNSTREAM PATH BEFORE
             ; COMPARING PATH LENGTHS.
```

```
                (t
                ; NEW PATH IS DOWNSTREAM OF OLD PATH.  ADD THE DIFFERENCE
                ; BETWEEN THE TWO TO THE NEW PAIR AND THEN COMPARE PATH
                  ;  LENGTHS.
                 (let ((distance-between-vertices
                         (straight-line-distance terminating-vertex
                                                  (landing-vertex (plus-mode ending-polygon))))
                        (adjusted-path-length-new-path (+ previous-polygon-cost
                                                           tangent-length
                                                           edge-traversal-cost))
                        (adjusted-path-length-old-path (cost (plus-mode ending-polygon))))
                  (cond ((< path-area (path-area (plus-mode ending-polygon)))
                        ; NEW PATH IS DOWNSTREAM OF OLD PATH.  ADD THE
                          ; DIFFERENCE
                         ; BETWEEN THE TWO TO THE NEW PATH AND
                          ; THEN COMPARE PATH LENGTHS.
                         (setf adjusted-path-length-new-path (+ adjusted-path-length-new-path
                                                                 distance-between-vertices)))
                        ((> path-area (path-area (plus-mode ending-polygon)))
                        ; NEW PATH IS UPSTREAM OF OLD PATH. ADD THE DIFFERENCE
                        ; BETWEEN THE TWO TO THE OLD PATH AND THEN COMPARE
                          ;  PATH LENGTHS.
                         (setf adjusted-path-length-old-path (+ adjusted-path-length-old-path
                                                                 distance-between-vertices))))
                    ; IF ADJUSTED PATH LENGTH OF NEW PATH IS LESS, THEN THE
                    ; NEW PATH IS THE BEST.  OTHERWISE, DO NOT CHANGE.
                  (cond ((< adjusted-path-length-new-path adjusted-path-length-old-path)
                         (erase-dark-line-3 *win-3*
                                            *my-bitmap-stream-3*
                                            (from-vertex (plus-mode ending-polygon))
                                            (landing-vertex (plus-mode ending-polygon)))
                         (build-plus-path-node originating-vertex
                                               terminating-vertex
                                               previous-polygon-path-node
                                               previous-polygon-cost
                                               previous-polygon-path-area
                                               edge-traversal-cost
                                               tangent-mode
                                               tangent-length
                                               ending-polygon
                                               path-area)))))))
        ;...................................................................
        ; LANDING MODE IS MINUS SO MUST BUILD A MINUS-MODE PATH-NODE
        ;...................................................................
        ((or (equal tangent-mode "minus-minus")
             (equal tangent-mode "plus-minus"))
           ; CURRENT POLYGON MINUS MODE NOT YET VISITED
```

```
(cond ((null (minus-mode ending-polygon))
      (build-minus-path-node originating-vertex
                                              terminating-vertex
                                              previous-polygon-path-node
                                              previous-polygon-cost
                                              previous-polygon-path-area
                                              edge-traversal-cost
                                              tangent-mode
                                              tangent-length
                                              ending-polygon
                                              path-area))
      ; CURRENT POLYGON MINUS MODE PREVIOUSLY VISITED USING
      ; THE SAME LANDING/TERMINATING VERTEX.  SIMPLY COMPARE
      ; PATH COST.  MINUS PATH NODE GETS THE VALUE OF THE
             ; PATH WITH THE LOWEST PATH COST.
             ((eq terminating-vertex
                 (landing-vertex (minus-mode ending-polygon)))
                    ; NEW PATH COST LESS THAN CURRENT PATH COST
              (cond ((< (+ previous-polygon-cost
                                tangent-length
                                edge-traversal-cost)
                              (cost (minus-mode ending-polygon)))
                          (erase-dark-line-3 *win-3*
                                                      *my-bitmap-stream-3*
                                                      (from-vertex (minus-mode ending-polygon))
                                                      (landing-vertex (minus-mode ending-polygon)))
                          (build-minus-path-node originating-vertex
                                                              terminating-vertex
                                                              previous-polygon-path-node
                                                              previous-polygon-cost
                                                              previous-polygon-path-area
                                                              edge-traversal-cost
                                                              tangent-mode
                                                              tangent-length
                                                              ending-polygon
                                                              path-area))))
      ; CURRENT POLYGON MINUS NODE PREVIOUSLY VISITED USING
      ; DIFFERENT LANDING/TERMINATING VERTICES. MUST CONSIDER
      ; PATH AREA TO DETERMINE WHICH PATH IS UPSTREAM/DOWNSTREAM.
      ; MUST ADD EDGE TRAVERSAL COST TO DOWNSTREAM PATH BEFORE
      ; COMPARING PATH LENGTHS.
             (t
             ; NEW PATH IS DOWNSTREAM OF OLD PATH.  ADD THE DIFFERENCE
             ; BETWEEN THE TWO TO THE NEW PAIR AND THEN COMPARE PATH
                ;  LENGTHS.
              (let ((distance-between-vertices
                          (straight-line-distance terminating-vertex
                                        (landing-vertex (minus-mode ending-polygon))))
                    (adjusted-path-length-new-path (+ previous-polygon-cost
                                                      tangent-length
                                                      edge-traversal-cost))
                    (adjusted-path-length-old-path (cost (minus-mode ending-polygon))))
                 (cond ((< path-area (path-area (minus-mode ending-polygon)))
                    ; NEW PATH IS DOWNSTREAM OF OLD PATH.  ADD THE
                        ; DIFFERENCE BETWEEN THE TWO TO THE NEW PATH
                        ;  AND THEN COMPARE PATH LENGTHS.
                      (setf adjusted-path-length-new-path (+ adjusted-path-length-new-path
                                                             distance-between-vertices)))
                     ((> path-area (path-area (minus-mode ending-polygon)))
                    ; NEW PATH IS UPSTREAM OF OLD PATH. ADD THE DIFFERENCE
                    ; BETWEEN THE TWO TO THE OLD PATH AND THEN COMPARE
                        ;  PATH LENGTHS.
                      (setf adjusted-path-length-old-path (+ adjusted-path-length-old-path
                                                             distance-between-vertices))))
                   ; IF ADJUSTED PATH LENGTH OF NEW PATH IS LESS, THEN THE
                   ; NEW PATH IS THE BEST.  OTHERWISE, DO NOT CHANGE.
                 (cond ((< adjusted-path-length-new-path adjusted-path-length-old-path)
                      (erase-dark-line-3 *win-3*
                                         *my-bitmap-stream-3*
                                         (from-vertex (minus-mode ending-polygon))
                                         (landing-vertex (minus-mode ending-polygon)))
                      (build-minus-path-node originating-vertex
                                             terminating-vertex
                                             previous-polygon-path-node
                                             previous-polygon-cost
                                             previous-polygon-path-area
                                             edge-traversal-cost
                                             tangent-mode
                                             tangent-length
                                             ending-polygon
                                             path-area))))))))))


;-------------------------------------------------------
;    DEFMETHOD BUILD-PLUS-PATH-NODE
;-------------------------------------------------------

(defmethod build-plus-path-node ((originating-vertex vertex)
                                 (terminating-vertex vertex)
                                 (previous-polygon-path-node path-node)
                                 previous-polygon-cost
                                 previous-polygon-path-area
                                 edge-traversal-cost
                                 tangent-mode
```

```
                         tangent-length
                         ending-polygon
                         path-area)
    (if (equalp *display* "yes")
       (draw-dark-line-3 *win-3*
                             *my-bitmap-stream-3*
                             originating-vertex
                             terminating-vertex))
    (setf (plus-mode ending-polygon)
     (make-path-node terminating-vertex
                     originating-vertex
                     ending-polygon
                     tangent-mode
                     (+ previous-polygon-cost
                       tangent-length
                       edge-traversal-cost)))
    ;----------- SETTING VALUES IN THE PATH NODE --------------------
    ; PATH-MODE
    (setf (path-mode (plus-mode ending-polygon)) "plus")
     ; SYMBOLIC PATH
    (setf (path (plus-mode ending-polygon))
     (reverse (cons (list (name ending-polygon) "+")
                    (reverse (path previous-polygon-path-node)))))
     ; DISTANCE SLOT
    (setf (distance (plus-mode ending-polygon))
     (straight-line-distance terminating-vertex *goal-point*))
     ; TOTAL-PATH-COST SLOT
    (setf (total-path-cost (plus-mode ending-polygon))
     (+ (cost (plus-mode ending-polygon))
       (distance (plus-mode ending-polygon))))
     ; PATH-AREA SLOT
    (setf (path-area (plus-mode ending-polygon))
     (+ path-area previous-polygon-path-area))
    ;-----------------------------------------------------------------
    (cond ((null *polygon-mode-list*)
           (setf *polygon-mode-list* (list (plus-mode ending-polygon))))
          (t (setf *polygon-mode-list*
              (cons (plus-mode ending-polygon) *polygon-mode-list*)))))

;----------------------------------------------------------
;    DEFMETHOD BUILD-MINUS-PATH-NODE
;----------------------------------------------------------
(defmethod build-minus-path-node ((originating-vertex vertex)
                          (terminating-vertex vertex)
                          (previous-polygon-path-node path-node)
                          previous-polygon-cost
                          previous-polygon-path-area
```

```
                        edge-traversal-cost
                        tangent-mode
                        tangent-length
                        ending-polygon
                        path-area)
(if (equalp *display* "yes")
   (draw-dark-line-3 *win-3*
                          *my-bitmap-stream-3*
                          originating-vertex
                          terminating-vertex))
(setf (minus-mode ending-polygon)
  (make-path-node terminating-vertex
                      originating-vertex
                      ending-polygon
                      tangent-mode
                      (+ previous-polygon-cost
                        tangent-length
                        edge-traversal-cost)))
 ;----------- SETTING VALUES IN THE PATH NODE -------------------
 ; PATH-MODE
(setf (path-mode (minus-mode ending-polygon)) "minus")
 ; SYMBOLIC PATH
(setf (path (minus-mode ending-polygon))
  (reverse (cons (list (name ending-polygon) "-")
                     (reverse (path previous-polygon-path-node)))))
 ; DISTANCE SLOT
(setf (distance (minus-mode ending-polygon))
  (straight-line-distance terminating-vertex *goal-point*))
 ; TOTAL-PATH-COST SLOT
(setf (total-path-cost (minus-mode ending-polygon))
  (+ (cost (minus-mode ending-polygon))
    (distance (minus-mode ending-polygon))))
 ; PATH-AREA SLOT
(setf (path-area (minus-mode ending-polygon))
  (+ path-area
    previous-polygon-path-area))
 ;---------------------------------------------------------------
(cond ((null *polygon-mode-list*)
         (setf *polygon-mode-list*
           (list (minus-mode ending-polygon))))
        (t (setf *polygon-mode-list*
             (cons (minus-mode ending-polygon)
                     *polygon-mode-list*)))))

;---------------------------------------------------------
;   DEFMETHOD COMPUTE-PATH-AREA
;---------------------------------------------------------
```

```
; S =   ((x1 - x-start) * (y2 - y-start))
;      - ((y1 - y-start) * (x2 - x-start))
;----------------------------------------------------------
(defmethod compute-path-area ((first-vertex  vertex)
                              (second-vertex vertex))
 (let ((x-start (x-coord (coordinates *start-point*)))
       (y-start (y-coord (coordinates *start-point*)))
       (x1     (x-coord (coordinates first-vertex)))
       (y1     (y-coord (coordinates first-vertex)))
       (x2     (x-coord (coordinates second-vertex)))
       (y2     (y-coord (coordinates second-vertex))))
  (- (* (- x1 x-start) (- y2 y-start))
    (* (- y1 y-start) (- x2 x-start)))))

;---------------------------------------------------,
;       LOWER-COST-P                                ;
;---------------------------------------------------;
(defmethod lower-cost-p ((path-segment-1 path-node)
                         (path-segment-2 path-node))
 (< (total-path-cost path-segment-1)
   (total-path-cost path-segment-2)))

;---------------------------------------------------;
;       LOWER-COST-P                                ;
;---------------------------------------------------;
(defmethod sort-and-expand-best-path ()
 (setf *polygon-mode-list* (sort *polygon-mode-list*
                                 #'(lambda (p1 p2)
                                     (lower-cost-p p1 p2))))
 (let* ((expansion-node (first *polygon-mode-list*)))
  (setf *polygon-mode-list* (rest *polygon-mode-list*))
  (expand-path-list expansion-node *my-list*)))

;---------------------------------------------------;
;       LOWER-COST-P                                ;
;---------------------------------------------------;
(defun shortest-paths ()
 (loop
  (when (or (equal *goal-found-flag* "true")(null *polygon-mode-list*))(return "DONE"))
  (sort-and-expand-best-path)))
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virgina 22304-6145	2
2.	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Dr. Yutaka Kanayama Code CS/Ka, Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	8
4.	Dr. Neil Rowe Code CS/Ro, Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
5.	MAJ Jerry A. Crane 3290 S. Duncan Rd. Bloomington, Indiana 47403	2